

## CHAPTER 12



### Understanding Controls

In this chapter, you will learn:

- What a control is in Java
- About classes whose instances represent controls in JavaFX
- About controls such as Label, Button, CheckBox, RadioButton, Hyperlink, ChoiceBox, ComboBox, ListView, ColorPicker, DatePicker, TextField, TextArea, and Menu
- How to style controls using a CSS
- How to use the FileChooser and DirectoryChooser dialogs

### What Is a Control?

JavaFX lets you create applications using GUI components. An application with a GUI performs three tasks:

- Accepts inputs from the user through input devices such as a keyboard or a mouse
- Processes the inputs (or takes actions based on the input)
- Displays outputs

The UI provides a means to exchange information in terms of input and output between an application and its users. Entering text using a keyboard, selecting a menu item using a mouse, clicking a button, or other actions are examples of providing input to a GUI application. The application displays outputs on a computer monitor using text, charts, dialog boxes, and so forth.

Users interact with a GUI application using graphical elements called *controls* or *wIDGETS*. Buttons, labels, text fields, text area, radio buttons, and check boxes are a few examples of controls. Devices like a keyboard, a mouse, and a touch screen are used to provide input to controls. Controls can also display output to the users. Controls generate events that indicate an occurrence of some kind of interaction between the user and the control. For example, pressing a button using a mouse

or a spacebar generates an action event indicating that the user has pressed the button.

JavaFX provides a rich set of easy-to-use controls. Controls are added to layout panes that position and size them. Layout panes were discussed in Chapter 10. This chapter discusses how to use the controls available in JavaFX.

Typically, the MVP pattern (discussed in Chapter 11) is used to develop a GUI application in JavaFX. MVP requires you to have at least three classes and place your business logic in a certain way and in certain classes. Generally, this bloats the application code, although for the right reason. This chapter will focus on the different types of controls, not on learning the MVP pattern. You will embed classes required for MVP patterns into one class to keep the code brief and save a lot of space in this book as well!

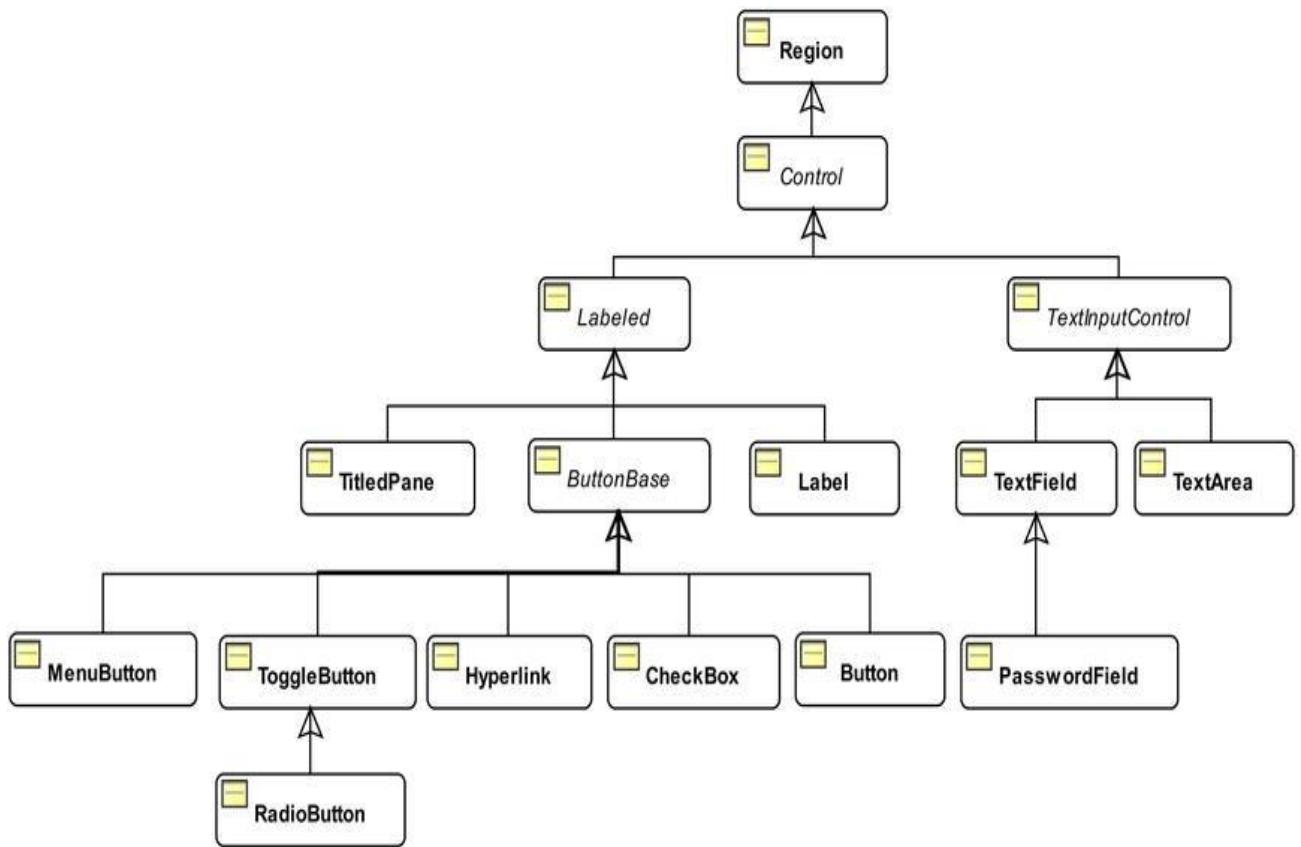
## Understanding Control Classes Hierarchy

Each control in JavaFX is represented by an instance of a class. If multiple controls share basic features, they inherit from a common base class. Control classes are included in the `javafx.scene.control` package. A control class is a subclass, direct or indirect, of the `Control` class, which in turn inherits from the `Region`. Recall that the `Region` class inherits from the `Parent` class. Therefore, technically, a `Control` is also a `Parent`. All our discussions about the `Parent` and `Region` classes in the previous chapters also apply to all control-related classes.

A `Parent` can have children. Typically, a control is composed of another node (sometimes, multiple nodes), which is its child node. Control classes do not expose the list of its children through the `getChildren()` method, and therefore, you cannot add any children to them.

Control classes expose the list of their internal unmodifiable children through the `getChildrenUnmodifiable()` method, which returns an `ObservableList<Node>`. You are not required to know about the internal children of a control to use the control. However, if you need the list of their children, the `getChildrenUnmodifiable()` method will give you that.

Figure 12-1 shows a class diagram for classes of some commonly used controls. The list of control classes is a lot bigger than the one shown in the class diagram.



**Figure 12-1.** A class diagram for control classes in JavaFX

The `Control` class is the base class for all controls. It declares three properties, as shown in Table 12-1, that are common to all controls.

**Table 12-1.** Properties Declared in the `Control` Class

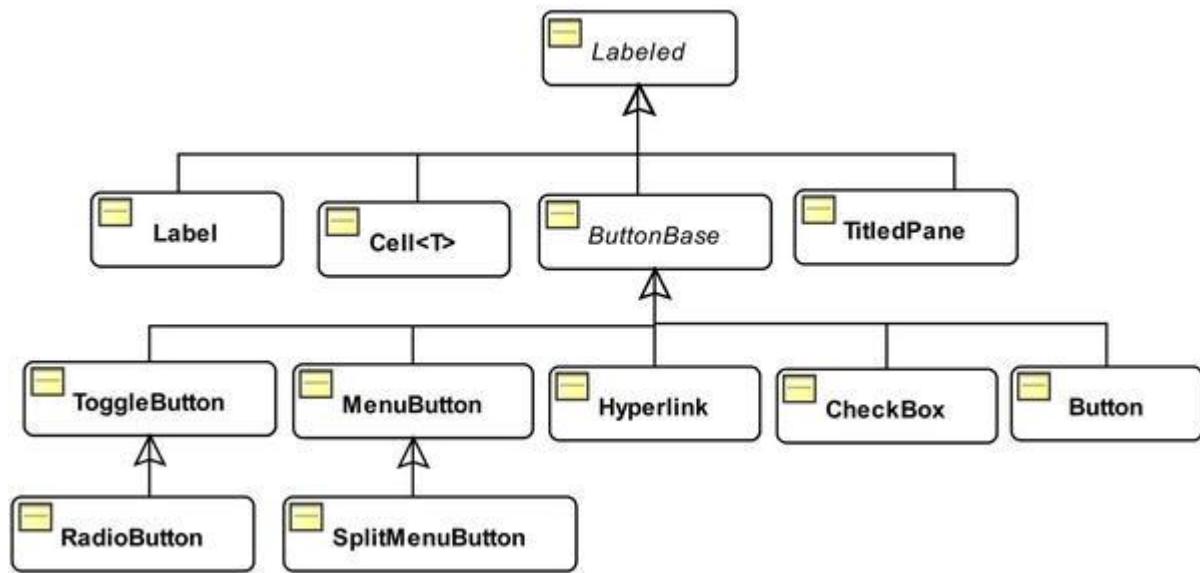
Property	Type	Description
contextMenu	<code>ObjectProperty&lt;ContextMenu&gt;</code>	Specifies the content menu for the control.
skin	<code>ObjectProperty&lt;Skin&lt;?&gt;&gt;</code>	Specifies the skin for the control.
tooltip	<code>ObjectProperty&lt;Tooltip&gt;</code>	Specifies the tool tip for the control.

The `contextMenu` property specifies the context menu for the control. A context menu gives a list of choices to the user. Each choice is an action that can be taken on the control in its current state. Some controls have their default context menus. For example, a `TextField`, when right-clicked, displays a context menu with choices like Undo, Cut, Copy, and Paste. Typically, a context menu is displayed when the user

Property	Type	Description
		presses a combination of keys (e.g., Shift + F10 on Windows) or clicks the mouse (right-click on Windows) when the control has focus. I will revisit the <code>contextMenu</code> property when I discuss the text input controls.
		At the time of this writing, JavaFX doesn't allow access or customization of the default context menu for controls. The <code>contextMenu</code> property is <code>null</code> even if the control has a default context menu. When you set the <code>contextMenu</code> property, it replaces the default context for the control. Note that not all controls have a default context menu and a context menu is not suitable for all controls. For example, a <code>Button</code> control does not use a context menu.
		The visual appearance of a control is known as its <i>skin</i> . A skin responds to the state changes in a control by changing its visual appearance. A skin is represented by an instance of the <code>Skin</code> interface. The <code>Control</code> class implements the <code>Skinnable</code> interface, giving all controls the ability to use a skin.
		The <code>skin</code> property in the <code>Control</code> class specifies the custom skin for a control. Developing a new skin is not an easy task. For the most part, you can customize the appearance of a control using CSS styles. All controls can be styled using CSS. The <code>Control</code> class implements the <code>Styleable</code> interface, so all controls can be styled. Please refer to Chapter 8 for more details on how to use a CSS. I will discuss some commonly used CSS attributes for some controls in this chapter.
		Controls can display a short message called a <i>tool tip</i> when the mouse hovers over the control for a short period. An object of the <code>Tooltip</code> class represents a tool tip in JavaFX. The <code>tooltip</code> property in the <code>Control</code> class specifies the tool tip for a control.
<h2>Labeled Controls</h2>		
		A labeled control contains a read-only textual content and optionally a graphic as part of its UI. <code>Label</code> , <code>Button</code> , <code>CheckBox</code> , <code>RadioButton</code> , and <code>Hyperlink</code> are some examples of labeled controls in JavaFX. All labeled controls are inherited, directly or indirectly, from the <code>Labeled</code> class, which is declared abstract.
		The <code>Labeled</code> class inherits from the <code>Control</code> class. Figure 12-2 shows a class diagram for labeled controls. Some of the classes have been left out in the diagram for brevity.

alignment	ObjectProperty<Pos>	It specifies the alignment of the content of the control within the content area. Its effect is visible when the content area is bigger than the content (text + graphic). The default value is Pos.CENTER_LEFT.
contentDisplay	ObjectProperty<ContentDisplay>	It specifies positioning of the graphic relative to the text.
ellipsisString	StringProperty	It specifies the string to display for the ellipsis when the text is truncated because the control has a smaller size than the preferred size. The default value is "..." for most locales. Specifying an empty string for this property does not display an ellipsis string in truncated text.
font	ObjectProperty<Font>	It specifies the default font for the text.
graphic	ObjectProperty<Node>	It specifies an optional icon for the control.
graphicTextGap	DoubleProperty	It specifies the amount of text between the graphic and text.
labelPadding	ReadOnlyObjectProperty<Insets>	It is the padding around the content area of the control. By default, it is Insets.EMPTY.
lineSpacing	DoubleProperty	It specifies the space between adjacent lines when the control displays multiple lines.

mnemonicParsing	BooleanProperty	It enables or disables text parsing to detect a mnemonic character. If it is set to true, the text for the control is parsed for an underscore (_) character. The character following the first underscore is added as the mnemonic for the control. Pressing the Alt key on Windows computers highlights mnemonics for all controls.
textAlignment	ObjectProperty<TextAlignment>	It specifies the text alignment within the text bounds for multiline text.
textFill	ObjectProperty<Paint>	It specifies the text color.
textOverrun	ObjectProperty<OverrunStyle>	It specifies how to display the text when the text content exceeds the available space.
text	StringProperty	It specifies the text content.
underline	BooleanProperty	It specifies whether the text content should be underlined.
wrapText	BooleanProperty	It specifies whether the text should be wrapped if the text cannot be displayed in one line.



**Figure 12-2.** A class diagram for labeled control classes

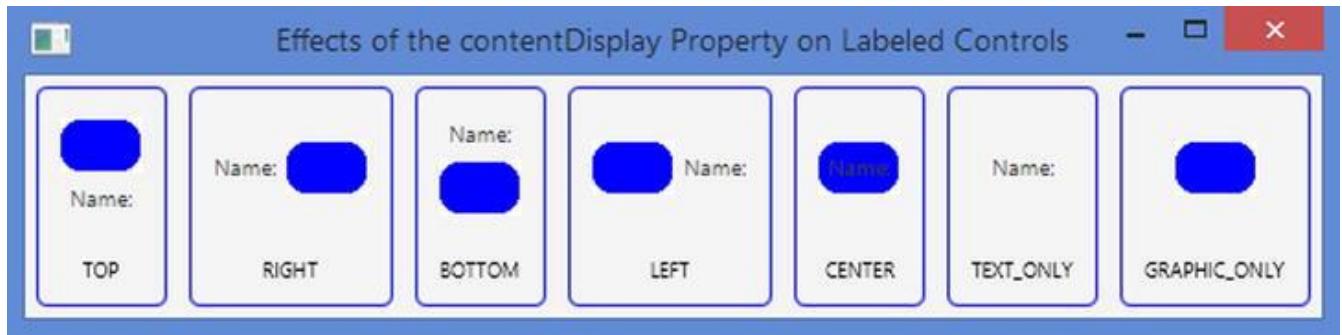
The `Labeled` class declares `text` and `graphic` properties to represent the textual and graphic contents, respectively. It declares several other properties to deal with the visual aspects of its contents, for example, alignment, font, padding, and text wrapping. Table 12-2 contains the list of those properties with their brief descriptions. I will discuss some of these properties in the subsequent sections.

**Table 12-2.** Properties Declared in the `Labeled` Class

### Positioning Graphic and Text

The `contentDisplay` property of labeled controls specifies the positioning of the graphic relative to the text. Its value is one of the constants of

the `ContentDisplay` enum: `TOP`, `RIGHT`, `BOTTOM`, `LEFT`, `CENTER`, `TEXT_ONLY`, and `GRAPHIC_ONLY`. If you do not want to display the text or the graphic, you can use the `GRAPHIC_ONLY` and `TEXT_ONLY` values instead of setting the text to an empty string and the graphic to `null`. Figure 12-3 shows the effects of using different values for the `contentDisplay` property of a `Label`. The `Label` uses `Name:` as the text and a blue rectangle as the graphic. The value for the `contentDisplay` property is displayed at the bottom of each instance.



**Figure 12-3.** Effects of the `contentDisplay` property on labeled controls

## Understanding Mnemonics and Accelerators

Labeled controls support keyboard *mnemonics*, which is also known as a *keyboard shortcut* or *keyboard indicator*. A mnemonic is a key that sends an `ActionEvent` to the control. The mnemonic key is often pressed in combination with a modifier key such as an Alt key. The modifier key is platform dependent; however, it is usually an Alt key. For example, suppose you set the C key as a mnemonic for a Close button. When you press Alt + C, the Close button is activated.

Finding the documentation about mnemonics in JavaFX is not easy. It is buried in the documentation for the `Labeled` and `Scene` classes. Setting a mnemonic key for a labeled control is easy. You need to precede the mnemonic character with an underscore in the text content and make sure that the `mnemonicParsing` property for the control is set to true. The first underscore is removed and the character following it is set as the mnemonic for the control. For some labeled controls, the mnemonic parsing is set to true by default, and for others, you will need to set it.

**Tip** Mnemonics are not supported on all platforms. Mnemonic characters in the text for controls are not underlined, at least on Windows, until the Alt key is pressed.

The following statement will set the C key as the mnemonic for the Close button:

```
// For Button, mnemonic parsing is true by default
Button closeBtn = new Button("_Close");
```

When you press the Alt key, the mnemonic characters for all controls are underlined and pressing the mnemonic character for any controls will set focus to the control and send it an `ActionEvent`.

JavaFX provides the following four classes in the `javafx.scene.input` package to set mnemonics for all types of controls programmatically:

- Mnemonic
- KeyCombination
- KeyCharacterCombination
- KeyCodeCombination

An object of the `Mnemonic` class represents a mnemonic. An object of the `KeyCombination` class, which is declared abstract, represents the key combination for a mnemonic.

The `KeyCharacterCombination` and `KeyCodeCombination` classes are subclasses of the `KeyCombination` class. Use the former to construct a key combination using a character; use the latter to construct a key combination using a key code. Note that not all keys on the keyboard represent characters. The `KeyCodeCombination` class lets you create a key combination for any key on the keyboard.

The `Mnemonic` object is created for a node and is added to a `Scene`. When the `Scene` receives an unconsumed key event for the key combination, it sends an `ActionEvent` to the target node.

The following snippet of code achieves the same result that was achieved using one statement in the above example:

```
Button closeBtn = new Button("Close");

// Create a KeyCombination for Alt + C
KeyCombination kc = new KeyCodeCombination(KeyCode.C,
KeyCombination.ALT_DOWN);

// Create a Mnemonic object for closeBtn
Mnemonic mnemonic = new Mnemonic(closeBtn, kc);

Scene scene = create a scene...
scene.addMnemonic(mnemonic); // Add the mnemonic to the scene
```

The `KeyCharacterCombination` class can also be used to create a key combination for Alt + C:

```
KeyCombination kc = new KeyCharacterCombination("C",
KeyCombination.ALT_DOWN);
```

The `Scene` class supports accelerator keys. An accelerator key, when pressed, executes a `Runnable`. Notice the difference between mnemonics and accelerator keys. A mnemonic is associated with a control, and pressing its key combination sends an `ActionEvent` to the control. An accelerator key is not associated with a control, but rather to a task. The `Scene` class maintains an `ObservableMap<KeyCombination, Runnable>`, whose reference can be obtained using the `getAccelerators()` method.

The following snippet of code adds an accelerator key (Ctrl + X on Windows and Meta + X on Mac) to a `Scene`, which closes the window

associated with the Scene. The SHORTCUT key represents the shortcut key on the platform—Ctrl on Windows and Meta on Mac:

```
Scene scene = create a scene object...;
...
KeyCombination kc = new KeyCodeCombination(KeyCode.X,
                                             KeyCombination.SHORTCUT_DOWN);
Runnable task = () -> scene.getWindow().hide();
scene.getAccelerators().put(kc, task);
```

The program in Listing 12-1 shows how to use mnemonics and accelerator keys. Press Alt + 1 and Alt + 2 to activate Button 1 and Button 2, respectively. Pressing these buttons changes the text for the Label. Pressing the shortcut key + X will close the window.

### ***Listing 12-1.*** Using Mnemonics and Accelerator Keys

```
// MnemonicTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.input.KeyCode;
import javafx.scene.input.KeyCodeCombination;
import javafx.scene.input.KeyCombination;
import javafx.scene.input.Mnemonic;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class MnemonicTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        VBox root = new VBox();
        root.setSpacing(10);
        root.setStyle("-fx-padding: 10;" +
                     "-fx-border-style: solid inside;" +
                     "-fx-border-width: 2;" +
                     "-fx-border-insets: 5;" +
                     "-fx-border-radius: 5;" +
                     "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        Label msg = new Label("Press Ctrl + X on Windows
\rand " +
                           "\nMeta + X on Mac to close the
window");
        Label lbl = new Label("Press Alt + 1 or Alt + 2");
```

```

        // Use Alt + 1 as the mnemonic for Button 1
        Button btn1 = new Button("Button _1");
        btn1.setOnAction(e -> lbl.setText("Button
1 clicked!"));

        // Use Alt + 2 as the mnemonic key for Button 2
        Button btn2 = new Button("Button 2");
        btn2.setOnAction(e -> lbl.setText("Button
2 clicked!"));

        KeyCombination kc = new
KeyCodeCombination(KeyCode.DIGIT2,
                               KeyCombination.ALT_
DOWN);

        Mnemonic mnemonic = new Mnemonic(btn2, kc);
        scene.addMnemonic(mnemonic);

        // Add an accelerator key to the scene
        KeyCombination kc4 =
            new KeyCodeCombination(KeyCode.X,
KeyCombination.SHORTCUT_DOWN);
        Runnable task = () -> scene.getWindow().hide();
        scene.getAccelerators().put(kc4, task);

        // Add all children to the VBox
        root.getChildren().addAll(msg, lbl, btn1, btn2);

        stage.setScene(scene);
        stage.setTitle("Using Mnemonics and Accelerators");
        stage.show();
    }
}

```

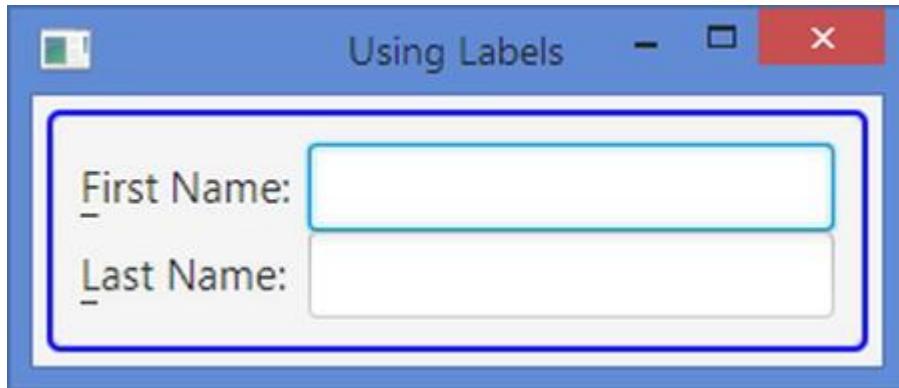
## Understanding the *Label* Control

An instance of the `Label` class represents a label control. As the name suggest, a `Label` is simply a label that is used to identify or describe another component on a screen. It can display a text, an icon, or both. Typically, a `Label` is placed next to (to the right or left) or at the top of the node it describes.

A `Label` is not focus traversable. That is, you cannot set the focus to a `Label` using the Tab key. A `Label` control does not generate any interesting events that are typically used in an application.

A `Label` control can also be used to display text in situations where it is acceptable to truncate the text if enough space is not available to display the entire text. Please refer to the API documentation on the `textOverrun` and `ellipsisString` properties of the `Labeled` class for more details on how to control the text truncation behavior in a `Label` control.

Figure 12-4 shows a window with two Label controls with text First Name: and Last Name:. The Label with the text First Name: is an indicator for the user that he should enter a first name in the field that is placed right next to it. A similar argument goes for the Last Name: Label control.



**Figure 12-4.** A window with two Label controls

The Label class has a very useful `labelFor` property of `ObjectProperty<Node>` type. It is set to another node in the scene graph. A Label control can have a mnemonic. Mnemonic parsing for Label controls is set to false by default. When you press the mnemonic key for a Label, the focus is set to the `labelFor` node for that Label. The following snippet of code creates a `TextField` and a `Label`. The `Label` sets a mnemonic, enables mnemonic parsing, and sets the `TextField` as its `labelFor` property. When the Alt + F keys are pressed, focus is moved to the `TextField`:

```
TextField fNameFld = new TextField();
Label fNameLbl = new Label("_First Name:");
fNameLbl.setLabelFor(fNameFld);
fNameLbl.setMnemonicParsing(true);
```

The program in Listing 12-2 produces the screen shown in Figure 12-4. Press Alt + F and Alt + L to shift focus between the two `TextField` controls.

### **Listing 12-2.** Using the Label Control

```
// LabelTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;
```

```
public class LabelTest extends Application {  
    public static void main(String[] args) {  
        Application.launch(args);  
    }  
  
    @Override  
    public void start(Stage stage) {  
        TextField fNameFld = new TextField();  
        Label fNameLbl = new Label("_First Name:");  
        fNameLbl.setLabelFor(fNameFld);  
        fNameLbl.setMnemonicParsing(true);  
  
        TextField lNameFld = new TextField();  
        Label lNameLbl = new Label("_Last Name:");  
        lNameLbl.setLabelFor(lNameFld);  
        lNameLbl.setMnemonicParsing(true);  
  
        GridPane root = new GridPane();  
        root.addRow(0, fNameLbl, fNameFld);  
        root.addRow(1, lNameLbl, lNameFld);  
        root.setStyle("-fx-padding: 10;" +  
                      "-fx-border-style: solid inside;" +  
                      "-fx-border-width: 2;" +  
                      "-fx-border-insets: 5;" +  
                      "-fx-border-radius: 5;" +  
                      "-fx-border-color: blue;");  
  
        Scene scene = new Scene(root);  
        stage.setScene(scene);  
        stage.setTitle("Using Labels");  
        stage.show();  
    }  
}
```

## Understanding Buttons

JavaFX provides three types of controls that represent buttons:

- Buttons to execute commands
- Buttons to make choices
- Buttons to execute commands as well as make choices

All button classes inherit from the `ButtonBase` class. Please refer to Figure 12-2 for a class diagram. All types of buttons support the `ActionEvent`. Buttons trigger an `ActionEvent` when they are activated. A button can be activated in different ways, for example, by using a mouse, a mnemonic, an accelerator key, or other key combinations.

A button that executes a command when activated is known as a *command button*. The `Button`, `Hyperlink`, and `MenuButton` classes

represent command buttons. A `MenuBar` lets the user execute a command from a list of commands. Buttons used for presenting different choices to users are known as *choice buttons*.

The `ToggleButton`, `CheckBox`, and `RadioButton` classes represent choice buttons. The third kind of button is a hybrid of the first two kinds. They let users execute a command or make choices.

The `SplitMenuItem` class represents a hybrid button.

**Tip** All buttons are labeled controls. Therefore, they can have a textual content, a graphic, or both. All types of buttons are capable of firing an `ActionEvent`.

## Understanding Command Buttons

You have already used command buttons in several instances, for example, a Close button to close a window. In this section, I will discuss buttons that are used as command buttons.

### Understanding the Button Control

An instance of the `Button` class represents a command button.

Typically, a `Button` has text as its label and an `ActionEvent` handler is registered to it. The `mnemonicParsing` property for the `Button` class is set to true by default.

A `Button` can be in one of three modes:

- A normal button
- A default button
- A cancel button

For a normal button, its `ActionEvent` is fired when the button is activated. For a default button, the `ActionEvent` is fired when the Enter key is pressed and no other node in the scene consumes the key press. For a cancel button, the `ActionEvent` is fired when the Esc key is pressed and no other node in the scene consumes the key press.

By default, a `Button` is a normal button. The default and cancel modes are represented by the `defaultButton` and `cancelButton` properties. You would set one of these properties to true to make a button a default or cancel button. By default, both properties are set to false.

The following snippet of code creates a normal `Button` and adds an `ActionEvent` handler. When the button is activated, for example, by clicking using a mouse, the `newDocument()` method is called:

```
// A normal button
Button newBtn = new Button("New");
newBtn.setOnAction(e -> newDocument());
```

The following snippet of code creates a default button and adds an `ActionEvent` handler. When the button is activated, the `save()` method is called. Note that a default `Button` is also activated by pressing the Enter key if no other node in the scene consumes the key press:

```
// A default button
Button saveBtn = new Button("Save");
saveBtn.setDefaultButton(true); // Make it a default button
saveBtn.setOnAction(e -> save());
```

The program in Listing 12-3 creates a normal button, a default button, and a cancel button. It adds an `ActionEvent` listener to all three buttons. Notice that all buttons have a mnemonic (e.g., N for the New button). When the buttons are activated, a message is displayed in a `Label`. You can activate the buttons by different means:

- Clicking on buttons
- Setting focus to the buttons using the Tab key and pressing the spacebar
- Pressing Alt key and their mnemonics
- Pressing the Enter key to activate the Save button
- Pressing Esc key to activate the Cancel button

No matter how you activate the buttons, their `ActionEvent` handler is called. Typically, the `ActionEvent` handler for a button contains the command for the button.

### ***Listing 12-3.*** Using the `Button` Class to Create Command Buttons

```
// ButtonTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ButtonTest extends Application {
    Label msgLbl = new Label("Press Enter or Esc key to see the
message");

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

```

    }

@Override
public void start(Stage stage) {
    // A normal button with N as its mnemonic
    Button newBtn = new Button("_New");
    newBtn.setOnAction(e -> newDocument());

    // A default button with S as its mnemonic
    Button saveBtn = new Button("_Save");
    saveBtn.setDefaultButton(true);
    saveBtn.setOnAction( e -> save());

    // A cancel button with C as its mnemonic
    Button cancelBtn = new Button("_Cancel");
    cancelBtn.setCancelButton(true);
    cancelBtn.setOnAction(e -> cancel());

    HBox buttonBox = new HBox(newBtn, saveBtn,
cancelBtn);
    buttonBox.setSpacing(15);
    VBox root = new VBox(msgLbl, buttonBox);
    root.setSpacing(15);
    root.setStyle("-fx-padding: 10;" +
"-fx-border-style: solid inside;" +
"-fx-border-width: 2;" +
"-fx-border-insets: 5;" +
"-fx-border-radius: 5;" +
"-fx-border-color: blue;");

    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("Command Buttons");
    stage.show();
}

public void newDocument() {
    msgLbl.setText("Creating a new document...");
}

public void save() {
    msgLbl.setText("Saving...");
}

public void cancel() {
    msgLbl.setText("Cancelling...");
}
}

```

**Tip** It is possible to set more than one button in a scene as a default or cancel button. However, only the first one is used. It is poor designing to declare multiple buttons as default and cancel buttons in a scene. By default, JavaFX highlights the default button with a light shade of color to give it a unique look. You can customize the appearance of default and cancel buttons using CSS

styles. Setting the same button as a default button and a cancel button is also allowed, but it is a sign of bad design when this is done.

The default CSS style-class name for a Button is `button`.

The `Button` class supports two CSS pseudo-classes: `default` and `cancel`. You can use these pseudo-classes to customize the look for default and cancel buttons. The following CSS style will set the text color for default buttons to blue and cancel buttons to gray:

```
.button:default {  
    -fx-text-fill: blue;  
}  
  
.button:cancel {  
    -fx-text-fill: gray;  
}
```

**Tip** You can use CSS styles to create stylish buttons. Please visit the web site at <http://fxexperience.com/2011/12/styling-fx-buttons-with-css/> for examples.

## Understanding the Hyperlink Control

An instance of the `Hyperlink` class represents a hyperlink control, which looks like a hyperlink in a web page. In a web page, a hyperlink is used to navigate to another web page. However, in JavaFX, an `ActionEvent` is triggered when a `Hyperlink` control is activated, for example, by clicking it, and you are free to perform any action in the `ActionEvent` handler.

A `Hyperlink` control is simply a button styled to look like a hyperlink. By default, mnemonic parsing is off. A `Hyperlink` control can have focus, and by default, it draws a dashed rectangular border when it has focus. When the mouse cursor hovers over a `Hyperlink` control, the cursor changes to a hand and its text is underlined.

The `Hyperlink` class contains a `visited` property of `BooleanProperty` type. When a `Hyperlink` control is activated for the first time, it is considered “visited” and the `visited` property is set to true automatically. All visited hyperlinks are shown in a different color than the not visited ones. You can also set the `visited` property manually using the `setVisited()` method of the `Hyperlink` class.

The following snippet of code creates a `Hyperlink` control with the text “JDojo” and adds an `ActionEvent` handler for the `Hyperlink`. When the `Hyperlink` is activated, the `www.jdojo.com` page is opened in a `WebView`, which is another JavaFX control to display a web

page. I will discuss the `WebView` control in Chapter 16; here I will use it without any explanation:

```
Hyperlink jdojoLink = new Hyperlink("JDojo");
WebView webview = new WebView();
jdojoLink.setOnAction(e ->
    webview.getEngine().load("http://www.jdojo.com"));
```

The program in Listing 12-4 adds three `Hyperlink` controls to the top region of a `BorderPane`. A `WebView` control is added in the center region. When you click one of the hyperlinks, the corresponding web page is displayed.

### ***Listing 12-4.*** Using the `Hyperlink` Control

```
// HyperlinkTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Hyperlink;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.web.WebView;
import javafx.stage.Stage;

public class HyperlinkTest extends Application {
    private WebView webview;

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

    @Override
    public void start(Stage stage) {
        // Must create a WebView object from the JavaFX
        Application Thread
        webview = new WebView();

        // Create some hyperlinks
        Hyperlink jdojoLink = new Hyperlink("JDojo");
        jdojoLink.setOnAction(e ->
            loadPage("http://www.jdojo.com"));

        Hyperlink yahooLink = new Hyperlink("Yahoo!");
        yahooLink.setOnAction(e ->
            loadPage("http://www.yahoo.com"));

        Hyperlink googleLink = new Hyperlink("Google");
        googleLink.setOnAction(e ->
            loadPage("http://www.google.com"));

        HBox linkBox = new HBox(jdojoLink, yahooLink,
```

```

googleLink);
linkBox.setSpacing(10);
linkBox.setAlignment(Pos.TOP_RIGHT);

BorderPane root = new BorderPane();
root.setTop(linkBox);
root.setCenter(webview);

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("Using Hyperlink Controls");
stage.show();
}

public void loadPage(String url) {
    webview.getEngine().load(url);
}
}
}

```

## Understanding the *MenuButton* Control

A *MenuButton* control looks like a button and behaves like a menu. When it is activated (by clicking or other means), it shows a list of options in the form of a pop-up menu. The list of options in the menu is maintained in an `ObservableList<MenuItem>` whose reference is returned by the `getItems()` method. To execute a command when a menu option is selected, you need to add the `ActionEventhandler` to the `MenuItem`s.

The following snippet of code creates a *MenuButton* with two *MenuItem*s. Each menu item has an `ActionEvent` hander attached to it. Figure 12-5 shows the *MenuButton* in two states: not showing and showing.

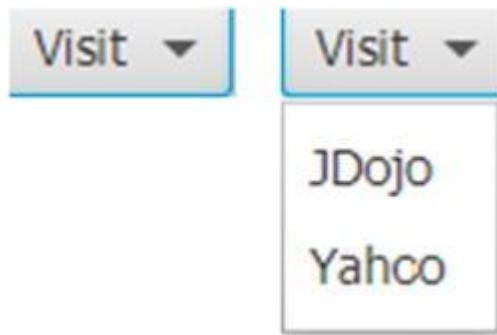
```

// Create two menu items with an ActionEvent handler.
// Assume that the loadPage() method exists
MenuItem jdojo = new MenuItem("JDojo");
jdojo.setOnAction(e -> loadPage("http://www.jdojo.com"));

MenuItem yahoo = new MenuItem("Yahoo");
yahoo.setOnAction(e -> loadPage("http://www.yahoo.com"));

// Create a MenuButton and the two menu items
MenuButton links = new MenuButton("Visit");
links.getItems().addAll(jdojo, yahoo);

```



**Figure 12-5.** A `MenuButton` in not showing and showing states

The `MenuButton` class declares two properties:

- `popupSide`
- `showing`

The `popupSide` property is of the `ObjectProperty<Side>` type and the `showing` property is of the `ReadOnlyBooleanProperty` type.

The `popupSide` property determines which side of the menu should be displayed. Its value is one of the constants in the `Side` enum: `TOP`, `LEFT`, `BOTTOM`, and `RIGHT`. The default value is `Side.BOTTOM`. An arrow in the `MenuItem` shows the direction set by the `popupSide` property. The arrow in Figure 12-5 is pointing downward, indicating that the `popupSide` property is set to `Side.BOTTOM`. The menu is opened in the direction set in the `popupSide` property only if space is available to display the menu in that side. If space is not available, the JavaFX runtime will make a smart decision as to which side the menu should be displayed. The value of the `showing` property is true when the pop-up menu is showing. Otherwise, it is false.

The program in Listing 12-5 creates an application using a `MenuButton` control that works similar to the one in Listing 12-4 that used `Hyperlink` control. Run the application, click the `Visit` `MenuButton` at the top right of the window, and select a page to open.

### **Listing 12-5.** Using the `MenuButton` Control

```
// MenuButtonTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
```

```
import javafx.scene.control.MenuButton;
import javafx.scene.control.MenuItem;
import javafx.scene.layout.BorderPane;
import javafx.scene.web.WebView;
import javafx.stage.Stage;

public class MenuButtonTest extends Application {
    private WebView webview;

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

    @Override
    public void start(Stage stage) {
        // Must create a WebView object from the JavaFX
        Application Thread
        webview = new WebView();

        MenuItem jdojo = new MenuItem("JDojo");
        jdojo.setOnAction(e ->
        loadPage("http://www.jdojo.com"));

        MenuItem yahoo = new MenuItem("Yahoo");
        yahoo.setOnAction(e ->
        loadPage("http://www.yahoo.com"));

        MenuItem google = new MenuItem("Google");
        google.setOnAction(e ->
        loadPage("http://www.google.com"));

        // Add menu items to the MenuButton
        MenuButton links = new MenuButton("Visit");
        links.getItems().addAll(jdojo, yahoo, google);

        BorderPane root = new BorderPane();
        root.setTop(links);
        BorderPane.setAlignment(links, Pos.TOP_RIGHT);
        root.setCenter(webview);

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using MenuButton Controls");
        stage.show();
    }

    public void loadPage(String url) {
        webview.getEngine().load(url);
    }
}
```

## Understanding Choice Buttons

JavaFX provides several controls to make one or more selections from a list of available choices:

- ToggleButton
- CheckBox
- RadioButton

**Tip** JavaFX also provides ChoiceBox, ComboBox, and ListView controls to allow the user to make a selection from multiple available choice. I will discuss these controls in a separate section.

All three controls are labeled controls and they help you present multiple choices to the user in different formats. The number of available choices may vary from two to N, where N is a number greater than two.

Selection from the available choices may be mutually exclusive. That is, the user can only make one selection from the list of choices. If the user changes the selection, the previous selection is automatically deselected. For example, the list of gender selection with three choices, Male, Female, and Unknown, is mutually exclusive. The user must select only one of the three choices, not two or more of them. The ToggleButton and RadioButton controls are typically used in this case.

There is a special case of selection where the number of choices is two. In this case, the choices are of boolean type: true or false. Sometimes, it is also referred to as a Yes/No or On/Off choice.

The ToggleButton and CheckBox controls are typically used in this case.

Sometimes the user can have multiple selections from a list of choices. For example, you may present the user with a list of hobbies to choose zero or more hobbies from the list.

The ToggleButton and CheckBox controls are typically used in this case.

### Understanding the *ToggleButton* Control

`ToggleButton` is a two-state button control. The two states are *selected* and *unselected*. Its `selected` property indicates whether it is selected. The `selected` property is true when it is in the selected state. Otherwise, it is false. When it is in the selected state, it stays depressed. You can toggle between the selected and unselected states by pressing it, and hence it got the name `ToggleButton`.

For `ToggleButtons`, mnemonic parsing is enabled by default.

Figure 12-6 shows four toggle buttons with Spring, Summer, Fall, and Winter as their labels. Two of the toggle buttons, Spring and Fall, are selected and the other two are unselected.



**Figure 12-6.** A window showing four toggle buttons

You create a `ToggleButton` the same way you create a `Button`, using the following code:

```
ToggleButton springBtn = new ToggleButton("Spring");
```

A `ToggleButton` is used to select a choice, not to execute a command. Typically, you do not add `ActionEvent` handlers to a `ToggleButton`. Sometimes you can use a `ToggleButton` to start or stop an action. For that, you will need to add a `ChangeListener` for its `selected` property.

**Tip** The `ActionEvent` handler for a `ToggleButton` is invoked every time you click it. Notice that the first click selects a `ToggleButton` and the second click deselects it. If you select and deselect a `ToggleButton`, the `ActionEvent` handler will be called twice.

Toggle buttons may be used in a group from which zero or one `ToggleButton` can be selected. To add toggle buttons to a group, you need to add them to a `ToggleGroup`. The `ToggleButton` class contains a `toggleGroup` property. To add a `ToggleButton` to a `ToggleGroup`, set the `toggleGroup` property of the `ToggleButton` to the group. Setting the `toggleGroup` property to null removes a `ToggleButton` from the group. The following snippet of code creates four toggle buttons and adds them to a `ToggleGroup`:

```
ToggleButton springBtn = new ToggleButton("Spring");
ToggleButton summerBtn = new ToggleButton("Summer");
ToggleButton fallBtn = new ToggleButton("Fall");
ToggleButton winterBtn = new ToggleButton("Winter");

// Create a ToggleGroup
ToggleGroup group = new ToggleGroup();

// Add all ToggleButtons to the ToggleGroup
springBtn.setToggleGroup(group);
summerBtn.setToggleGroup(group);
```

```
fallBtn.setToggleGroup(group);
winterBtn.setToggleGroup(group);
```

Each `ToggleGroup` maintains an `ObservableList<Toggle>`. Note that `Toggle` is an interface that is implemented by the `ToggleButton` class. The `getToggles()` method of the `ToggleGroup` class returns the list of `Toggles` in the group. You can add a `ToggleButton` to a group by adding it to the list returned by the `getToggles()` method. The above snippet of code may be rewritten as follows:

```
ToggleButton springBtn = new ToggleButton("Spring");
ToggleButton summerBtn = new ToggleButton("Summer");
ToggleButton fallBtn = new ToggleButton("Fall");
ToggleButton winterBtn = new ToggleButton("Winter");

// Create a ToggleGroup
ToggleGroup group = new ToggleGroup();

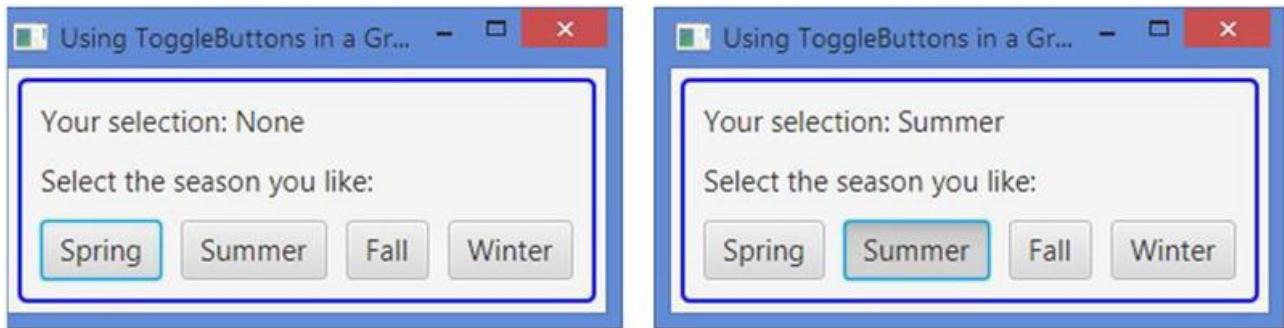
// Add all ToggleButtons to the ToggleGroup
group.getToggles().addAll(springBtn, summerBtn, fallBtn,
winterBtn);
```

The `ToggleGroup` class contains a `selectedToggle` property that keeps track of the selected `Toggle` in the group.

The `getSelectedToggle()` method returns the reference of the `Toggle` that is selected. If no `Toggle` is selected in the group, it returns `null`. Add a `ChangeListener` to this property if you are interested in tracking the change in selection inside a `ToggleGroup`.

**Tip** You can select zero or one `ToggleButton` in a `ToggleGroup`. Selecting a `ToggleButton` in a group deselects the already selected `ToggleButton`. Clicking an already selected `ToggleButton` in a group deselects it, leaving no `ToggleButton` in the group selected.

The program in Listing 12-6 adds four toggle buttons to a `ToggleGroup`. You can select none or at the most one `ToggleButton` from the group. Figure 12-7 shows two screenshots: one when there is no selection and one when the `ToggleButton` with the label `Summer` is selected. The program adds a `ChangeListener` to the group to track the change in selection and displays the label of the selected `ToggleButton` in a `Label` control.



**Figure 12-7.** Four toggle buttons in a `ToggleGroup` allowing selection of one button at a time

### **Listing 12-6.** Using Toggle Buttons in a `ToggleGroup` and Tracking the Selection

```
// ToggleButtonTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.beans.value.ObservableValue;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.Labeled;
import javafx.scene.control.Toggle;
import javafx.scene.control.ToggleButton;
import javafx.scene.control.ToggleGroup;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ToggleButtonTest extends Application {
    Label userSelectionMsg = new Label("Your selection: None");

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

    @Override
    public void start(Stage stage) {
        // Create four ToggleButtons
        ToggleButton springBtn = new ToggleButton("Spring");
        ToggleButton summerBtn = new ToggleButton("Summer");
        ToggleButton fallBtn = new ToggleButton("Fall");
        ToggleButton winterBtn = new ToggleButton("Winter");

        // Add all ToggleButtons to a ToggleGroup
        ToggleGroup group = new ToggleGroup();
        group.getToggles().addAll(springBtn, summerBtn,
        fallBtn, winterBtn);

        // Track the selection changes and display the
        // currently selected season
        group.selectedToggleProperty().addListener(this::cha
```

```

nged);

        Label msg = new Label("Select the season you
like:");

        // Add ToggleButtons to an HBox
        HBox buttonBox = new HBox(springBtn, summerBtn,
fallBtn, winterBtn);
        buttonBox.setSpacing(10);

        VBox root = new VBox(userSelectionMsg, msg,
buttonBox);
        root.setSpacing(10);
        root.setStyle("-fx-padding: 10;" +
                  "-fx-border-style: solid inside;" +
                  "-fx-border-width: 2;" +
                  "-fx-border-insets: 5;" +
                  "-fx-border-radius: 5;" +
                  "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using ToggleButtons in a Group");
        stage.show();
    }

    // A change listener to track the selection in the group
    public void changed(ObservableValue<? extends Toggle>
observable,
                      Toggle oldBtn,
                      Toggle newBtn) {
        String selectedLabel = "None";
        if (newBtn != null) {
            selectedLabel = ((Labeled)newBtn).getText();
        }

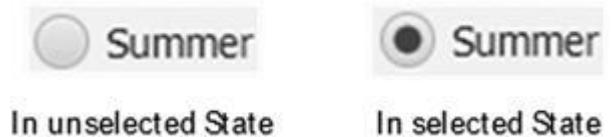
        userSelectionMsg.setText("Your selection: "
+ selectedLabel);
    }
}

```

## Understanding the *RadioButton* Control

An instance of the `RadioButton` class represents a radio button. It inherits from the `ToggleButton` class. Therefore, it has all of the features of a toggle button. A radio button is rendered differently compared to a toggle button. Like a toggle button, a radio button can be in one of the two states: *selected* and *unselected*. Its `selected` property indicates its current state. Like a toggle button, its mnemonic parsing is enabled by default. Like a toggle button, it also sends an `ActionEvent` when it is selected and unselected. Figure 12-8 shows

a RadioButton with Summer as its text in selected and unselected states.



*Figure 12-8. Showing a radio button in selected and unselected states*

There is a significant difference in the use of radio buttons compared to the use of toggle buttons. Recall that when toggle buttons are used in a group, there may not be any selected toggle button in the group. When radio buttons are used in a group, there must be one selected radio button in the group. Unlike a toggle button, clicking a selected radio button in a group does not unselect it. To enforce the rule that one radio button must be selected in a group of radio buttons, one radio button from the group is selected programmatically by default.

**Tip** Radio buttons are used when the user must make a selection from a list of choices. Toggle buttons are used when the user has an option to make one selection or no selection from a list of choices.

The program in Listing 12-7 shows how to use radio buttons inside a ToggleGroup. Figure 12-9 shows the window with the results of running the code. The program is very similar to the previous program that used toggle buttons. With the following code, Summer is set as the default selection:

```
// Select the default season as Summer
summerBtn.setSelected(true);
```

You set the default season in the radio button after you have added the change listener to the group, so the message to display the selected season is updated correctly.

### ***Listing 12-7.*** Using Radio Buttons in a ToggleGroup and Tracking the Selection

```
// RadioButtonTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.beans.value.ObservableValue;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.Labeled;
import javafx.scene.control.Toggle;
import javafx.scene.control.RadioButton;
import javafx.scene.control.ToggleGroup;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
```

```

import javafx.stage.Stage;

public class RadioButtonTest extends Application {
    Label userSelectionMsg = new Label("Your selection: None");

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

    @Override
    public void start(Stage stage) {
        // Create four RadioButtons
        RadioButton springBtn = new RadioButton("Spring");
        RadioButton summerBtn = new RadioButton("Summer");
        RadioButton fallBtn = new RadioButton("Fall");
        RadioButton winterBtn = new RadioButton("Winter");

        // Add all RadioButtons to a ToggleGroup
        ToggleGroup group = new ToggleGroup();
        group.getToggles().addAll(springBtn, summerBtn,
        fallBtn, winterBtn);

        // Track the selection changes and display the
        currently selected season
        group.selectedToggleProperty().addListener(this::cha
        nged);

        // Select the default season as Summer
        summerBtn.setSelected(true);

        Label msg = new Label("Select the season you like
        the most:");

        // Add RadioButtons to an HBox
        HBox buttonBox = new HBox(springBtn, summerBtn,
        fallBtn, winterBtn);
        buttonBox.setSpacing(10);

        VBox root = new VBox(userSelectionMsg, msg,
        buttonBox);
        root.setSpacing(10);
        root.setStyle("-fx-padding: 10;" +
                      "-fx-border-style: solid inside;" +
                      "-fx-border-width: 2;" +
                      "-fx-border-insets: 5;" +
                      "-fx-border-radius: 5;" +
                      "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using RadioButtons in a Group");
        stage.show();
    }

    // A change listener to track the selection in the group

```

```

public void changed(ObservableValue<? extends Toggle>
observable,
                     Toggle oldBtn,
                     Toggle newBtn) {
    String selectedLabel = "None";
    if (newBtn != null) {
        selectedLabel = ((Labeled) newBtn).getText();
    }
    userSelectionMsg.setText("Your selection: "
+ selectedLabel);
}
}

```

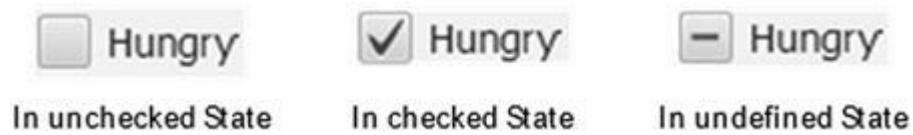


**Figure 12-9.** Four radio buttons in a *ToggleGroup*

### Understanding the *CheckBox* Control

*CheckBox* is a three-state selection control: *checked*, *unchecked*, and *undefined*. The *undefined* state is also known as an *indeterminate* state. A *CheckBox* supports a selection of three choices: true/false/unknown or yes/no/unknown. Usually, a *CheckBox* has text as a label, but not a graphic (even though it can). Clicking a *CheckBox* transitions it from one state to another cycling through three states.

A box is drawn for a *CheckBox*. In the *unchecked* state, the box is empty. A tick mark (or a check mark) is present in the box when it is in the *checked* state. In the *undefined* state, a horizontal line is present in the box. Figure 12-10 shows a *CheckBox* labeled *Hungry* in its three states.



**Figure 12-10.** Showing a check box in *unchecked*, *checked*, and *undefined* states

By default, the CheckBox control supports only two states: *checked* and *unchecked*. The `allowIndeterminate` property specifies whether the third state (the undefined state) is available for selection. By default, it is set to false:

```
// Create a CheckBox that supports checked and unchecked states only
CheckBox hungryCbx = new CheckBox("Hungry");

// Create a CheckBox and configure it to support three states
CheckBox agreeCbx = new CheckBox("Hungry");
agreeCbx.setAllowIndeterminate(true);
```

### The CheckBox class

contains `selected` and `indeterminate` properties to track its three states. If the `indeterminate` property is true, it is in the undefined state. If the `indeterminate` property is false, it is defined and it could be in a checked or unchecked state. If the `indeterminate` property is false and the `selected` property is true, it is in a checked state. If the `indeterminate` property is false and the `selected` property is false, it is in an unchecked state. Table 12-3 summarizes the rules for determining the state of a check box.

**Table 12-3.** Determining the State of a Check Box Based on Its Indeterminate and Selected Properties

indeterminate	selected	State
false	true	Checked
false	false	Unchecked
true	true/false	Undefined

Sometimes you may want to detect the state transition in a check box. Because a check box maintains the state information in two properties, you will need to add a `ChangeListener` to both properties.

An `ActionEvent` is fired when a check box is clicked. You can also use an `ActionEvent` to detect a state change in a check box. The following snippet of code shows how to use two `ChangeListeners` to detect a state change in a `CheckBox`. It is assumed that the `changed()` method and the rest of the code are part of the same class:

```
// Create a CheckBox to support three states
CheckBox agreeCbx = new CheckBox("I agree");
```

```

agreeCbx.setAllowIndeterminate(true);

// Add a ChangeListener to the selected and indeterminate
properties
agreeCbx.selectedProperty().addListener(this::changed);
agreeCbx.indeterminateProperty().addListener(this::changed);
...
// A change listener to track the selection in the group
public void changed(ObservableValue<? extends Boolean>
observable,
                     Boolean oldValue,
                     Boolean newValue) {
    String state = null;
    if (agreeCbx.isIndeterminate()) {
        state = "Undefined";
    } else if (agreeCbx.isSelected()) {
        state = "Checked";
    } else {
        state = "Unchecked";
    }
    System.out.println(state);
}
}

```

The program in Listing 12-8 shows how to use CheckBox controls. Figure 12-11 shows the window that results from running this code. The program creates two CheckBox controls. The Hungry CheckBox supports only two states. The I agree CheckBox is configured to support three states. When you change the state for the I agree CheckBox by clicking it, the Label at the top displays the description of the state.

### ***Listing 12-8.*** Using the CheckBox Control

```

// CheckBoxTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.beans.value.ObservableValue;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.CheckBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class CheckBoxTest extends Application {
    Label userSelectionMsg = new Label("Do you agree? No");
    CheckBox agreeCbx;

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

    @Override

```

```

public void start(Stage stage) {
    // Create a CheckBox to support only two states
    CheckBox hungryCbx = new CheckBox("Hungry");

    // Create a CheckBox to support three states
    agreeCbx = new CheckBox("I agree");
    agreeCbx.setAllowIndeterminate(true);

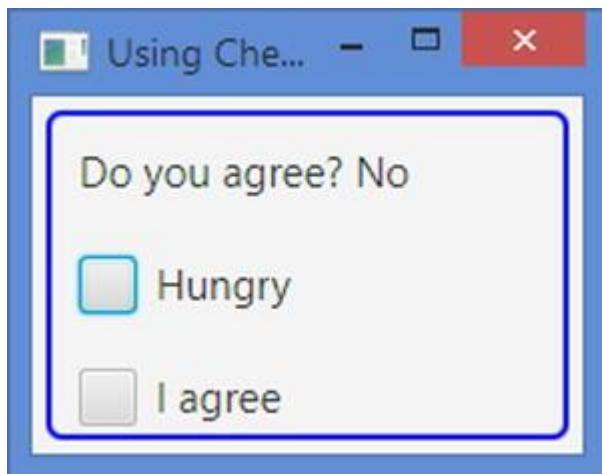
    // Track the state change for the "I agree" CheckBox
    // Text for the Label userSelectionMsg will be
    updated
    agreeCbx.selectedProperty().addListener(this::change
d);
    agreeCbx.indeterminateProperty().addListener(this::c
hanged);

    VBox root = new VBox(userSelectionMsg, hungryCbx,
    agreeCbx);
    root.setSpacing(20);
    root.setStyle("-fx-padding: 10;" +
                  "-fx-border-style: solid inside;" +
                  "-fx-border-width: 2;" +
                  "-fx-border-insets: 5;" +
                  "-fx-border-radius: 5;" +
                  "-fx-border-color: blue;");

    Scene scene = new Scene(root, 200, 130);
    stage.setScene(scene);
    stage.setTitle("Using CheckBoxes");
    stage.show();
}

// A change listener to track the state change in agreeCbx
public void changed(ObservableValue<? extends Boolean>
observable,
                     Boolean oldValue,
                     Boolean newValue) {
    String msg;
    if (agreeCbx.isIndeterminate()) {
        msg = "Not sure";
    } else if (agreeCbx.isSelected()) {
        msg = "Yes";
    } else {
        msg = "No";
    }
    this.userSelectionMsg.setText("Do you agree? "
+ msg);
}
}
}

```



**Figure 12-11.** Two check boxes: one uses two states and one uses three states

The default CSS style-class name for a CheckBox is `check-box`. The CheckBox class supports three CSS pseudo-classes: `selected`, `determinate`, and `indeterminate`. The `selected` pseudo-class applies when the `selected` property is true. The `determinate` pseudo-class applies when the `indeterminate` property is false. The `indeterminate` pseudo-class applies when the `indeterminate` property is true.

The CheckBox control contains two substructures: `box` and `mark`. You can style them to change their appearance. You can change the background color and border for the box and you can change the color and shape of the tick mark. Both `box` and `mark` are an instance of StackPane. The tick mark is shown giving a shape to the StackPane. You can change the shape for the mark by supplying a different shape in a CSS. By changing the background color of the mark, you change the color of the tick mark. The following CSS will show the box in tan and tick mark in red:

```
.check-box .box {
    -fx-background-color: tan;
}

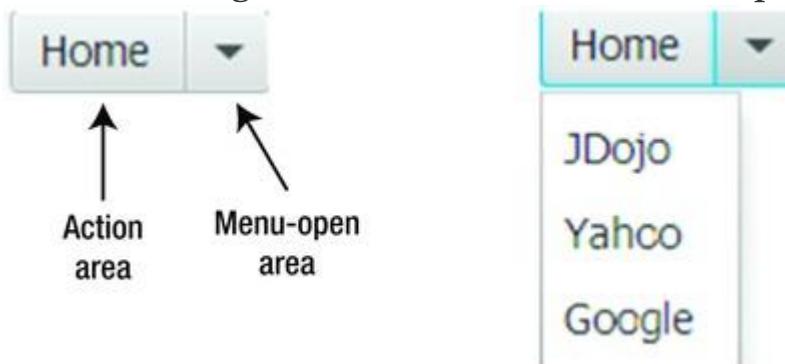
.check-box:selected .mark {
    -fx-background-color: red;
}
```

## Understanding the Hybrid **Button** Control

With our definitions of different button types, a SplitMenuItem falls under the hybrid category. It combines the features of a pop-up menu and a command button. It lets you select an action like a MenuButton control and execute a command like a Button control. The SplitMenuItem class inherits from the MenuItem class.

A `SplitMenuItem` is divided into two areas: the action area and the menu-open area. When you click in the action area, `ActionEvent` is fired. The registered `ActionEvent` handlers execute the command. When the menu-open area is clicked, a menu is shown from which the user will select an action to execute. Mnemonic parsing for `SplitMenuItem` is enabled by default.

Figure 12-12 shows a `SplitMenuItem` in two states. The picture on the left shows it in the collapsed state. In the picture on the right, it shows the menu items. Notice the vertical line dividing the control in two halves. The half containing the text Home is the action area. The other half containing the down arrow is the menu-open area.



**Figure 12-12.** A `SplitMenuItem` in the collapsed and showing states

You can create a `SplitMenuItem` with menu items or without them using its constructors with the following code:

```
// Create an empty SplitMenuItem
SplitMenuItem splitBtn = new SplitMenuItem();
splitBtn.setText("Home"); // Set the text as "Home"

// Create MenuItem
MenuItem jdojo = new MenuItem("JDojo");
MenuItem yahoo = new MenuItem("Yahoo");
MenuItem google = new MenuItem("Google");

// Add menu items to the MenuButton
splitBtn.getItems().addAll(jdojo, yahoo, google);
```

You need to add an `ActionEvent` handler to execute an action when the `SplitMenuItem` is clicked in the action area:

```
// Add ActionEvent handler when "Home" is clicked
splitBtn.setOnAction(e -> /* Take some action here */);
```

The program in Listing 12-9 shows how to use a `SplitMenuItem`. It adds a `SplitMenuItem` with the text Home and three menu items in the top right region of a `BorderPane`. A `WebView` is added in the center region. When you click Home, the `www.jdojo.com` web page is opened. When you select a web site using the menu by clicking the down

arrow, the corresponding web site is opened. The program is very similar to the ones you developed earlier using `MenuButton` and `Hyperlink` controls.

### ***Listing 12-9.*** Using the `SplitMenuItem` Control

```
// SplitMenuItemTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.MenuItem;
import javafx.scene.control.SplitMenuItem;
import javafx.scene.layout.BorderPane;
import javafx.scene.web.WebView;
import javafx.stage.Stage;

public class SplitMenuItemTest extends Application {
    private WebView webview;

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

    @Override
    public void start(Stage stage) {
        // Must create a WebView object from the JavaFX
        Application Thread
        webview = new WebView();

        MenuItem jdojo = new MenuItem("JDojo");
        jdojo.setOnAction(e ->
loadPage("http://www.jdojo.com"));

        MenuItem yahoo = new MenuItem("Yahoo");
        yahoo.setOnAction(e ->
loadPage("http://www.yahoo.com"));

        MenuItem google = new MenuItem("Google");
        google.setOnAction(e ->
loadPage("http://www.google.com"));

        // Create a SplitMenuItem
        SplitMenuItem splitBtn = new SplitMenuItem();
        splitBtn.setText("Home");

        // Add menu items to the SplitMenuItem
        splitBtn.getItems().addAll(jdojo, yahoo, google);

        // Add ActionEvent handler when "Home" is clicked
        splitBtn.setOnAction(e ->
loadPage("http://www.jdojo.com"));
    }
}
```

```

        BorderPane root = new BorderPane();
        root.setTop(splitBtn);
        BorderPane.setAlignment(splitBtn, Pos.TOP_RIGHT);
        root.setCenter(webview);

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using SplitMenuItem Controls");
        stage.show();
    }

    public void loadPage(String url) {
        webview.getEngine().load(url);
    }
}

```

## Making Selections from a List of Items

In the previous sections, you have seen how to present users with a list of items, for example, using toggle buttons and radio buttons. Toggle and radio buttons are easier to use because all options are always visible to the users. However, they use a lot of space on the screen. Think about using radio buttons to show the names of all 50 states in the United States to the user. It would take a lot of space. Sometimes none of the available items in the list is suitable for selection, so you will want to give users a chance to enter a new item that is not in the list.

JavaFX provides some controls that let users select an item(s) from a list of items. They take less space compared to buttons. They provide advanced features to customize their appearance and behaviors. I will discuss the following such controls in subsequent sections:

- ChoiceBox
- ComboBox
- ListView
- ColorPicker
- DatePicker

`ChoiceBox` lets users select an item from a small list of predefined items. `ComboBox` is an advanced version of `ChoiceBox`. It has many features, for example, the ability to be editable or change the appearance of the items in the list, which are not offered in `ChoiceBox`.

`ListView` provides users an ability to select multiple items from a list of items. Typically, all or more than one item in a `ListView` is visible to the user all the time. `ColorPicker` lets users select a color from a standard color palette or define a custom color graphically. `DatePicker` lets users select a date from a calendar pop-up. Optionally, users can enter a

date as text. ComboBox, ColorPicker, and DatePicker have the same superclass ComboBoxBase.

## Understanding the **ChoiceBox** Control

ChoiceBox is used to let a user select an item from a small list of items. The items may be any type of objects. ChoiceBox is a parameterized class. The parameter type is the type of the items in its list. If you want to store mixed types of items in a ChoiceBox, you can use its raw type, as shown in the following code:

```
// Create a ChoiceBox for any type of items
ChoiceBox seasons = new ChoiceBox();

// Create a ChoiceBox for String items
ChoiceBox<String> seasons = new ChoiceBox<String>();
```

You can specify the list items while creating a ChoiceBox with the following code:

```
ObservableList<String> seasonList
= FXCollections.<String>observableArrayList(
        "Spring", "Summer", "Fall",
        "Winter");
ChoiceBox<String> seasons = new ChoiceBox<>(seasonList);
```

After you create a ChoiceBox, you can add items to its list of items using the items property, which is of the ObjectProperty<ObservableList<T>> type in which T is the type parameter for the ChoiceBox. The following code will accomplish this:

```
ChoiceBox<String> seasons = new ChoiceBox<>();
seasons.getItems().addAll("Spring", "Summer", "Fall", "Winter");
```

Figure 12-13 shows a choice box in four different states. It has four names of seasons in the list of items. The first picture (labeled #1) shows it in its initial state when there is no selection. The user can open the list of items using the mouse or the keyboard. Clicking anywhere inside the control opens the list of items in a pop-up window, as shown in the picture labeled #2. Pressing the down arrow key when the control has focus also opens the list of items. You can select an item from the list by clicking it or using the up/down arrow and the Enter key. When you select an item, the pop-up window showing the items list is collapsed and the selected item is shown in the control, as shown in the picture labeled #3. The picture labeled #4 shows the control when an item is selected (Spring in this case) and the list items are shown. The pop-up window displays a check mark with the item already selected in the control. Table 12-4 lists the properties declared in the ChoiceBox class.



**Figure 12-13.** A choice box in different states

**Table 12-4.** Properties Declared in the ChoiceBox Class

Property	Type	Description
converter	ObjectProperty<StringConverter<T>>	It serves as a converter object whose <code>toString()</code> method is used to get the string representation of the items in the list.
items	ObjectProperty<ObservableList<T>>	It is the list of choices to display in the ChoiceBox.
selectionModel	ObjectProperty<SingleSelectionModel<T>>	It serves as a selection model to track of the selections in a ChoiceBox.
showing	ReadOnlyBooleanProperty	Its true value indicates that showing the list of choices. Its false value indicates that the choices is collapsed.
value	ObjectProperty<T>	It is the selected item in the ChoiceBox.

**Tip** You are not limited to showing the items list using the mouse or keyboard. You can show and hide the list programmatically using the `show()` and `hide()` methods, respectively.

The `value` property of the `ChoiceBox` stores the selected item in the control. Its type is `ObjectProperty<T>`, where `T` is the type parameter for the control. If the user has not selected an item, its value is `null`. The following snippet of code sets the `value` property:

```
// Create a ChoiceBox for String items
ChoiceBox<String> seasons = new ChoiceBox<String>();
seasons.getItems().addAll("Spring", "Summer", "Fall", "Winter");

// Get the selected value
String selectedValue = seasons.getValue();

// Set a new value
seasons.setValue("Fall");
```

When you set a new value using the `setValue()` method, the `ChoiceBox` selects the specified value in the control if the value exists in the list of items. It is possible to set a value that does not exist in the list of items. In that case, the `value` property contains the newly set item, but the control does not show it. The control keeps showing the previously selected item, if any. When the new item is later added to the list of items, the control shows the item set in the `value` property.

The `ChoiceBox` needs to track the selected item and its index in the list of items. It uses a separate object, called the *selection model*, for this purpose. The `ChoiceBox` class contains a `selectionModel` property to store the item selection details. `ChoiceBox` uses an object of the `SingleSelectionModel` class as its selection model, but you can use your own selection model. The default selection model works in almost all cases. The selection model provides you selection-related functionality:

- It lets you select an item using the index of the item in the list.
- It lets you select the first, next, previous, or last item in the list.
- It lets you clear the selection.
- Its `selectedIndex` and `selectedItem` properties track the index and value of the selected item. You can add a `ChangeListener` to these properties to handle a change in selection in a `ChoiceBox`. When no item is selected, the selected index is `-1` and the selected item is `null`.

The following snippet of code forces a value in a `ChoiceBox` by selecting the first item in the list by default:

```
ChoiceBox<String> seasons = new ChoiceBox<>();
seasons.getItems().addAll("Spring", "Summer", "Fall", "Winter",
"Fall");

// Select the first item in the list
seasons.getSelectionModel().selectFirst();
```

Use the `selectNext()` method of the selection model to select the next item from the list. Calling the `selectNext()` method when the last item is already selected has no effect. Use the `selectPrevious()` and `selectLast()` methods to select the previous and the last item in the list, respectively. The `select(int index)` and `select(T item)` methods select an item using the index and value of the item, respectively. Note that you can also use the `setValue()` method of the `ChoiceBox` to select an item from the list by its value. The `clearSelection()` method of the selection model clears the current selection, returning the `ChoiceBox` to a state as if no item had been selected.

The program in Listing 12-10 displays a window as shown in Figure 12-14. It uses a `ChoiceBox` with a list of four seasons. By default, the program selects the first season from the list. The application forces the user to select one season name by selecting one by default. It adds `ChangeListeners` to the `selectedIndex` and `selectedItem` properties of the selection model. They print the details of the selection change on the standard output. The current selection is shown in a `Label` control whose `text` property is bound to the `value` property of the `ChoiceBox`. Select a different item from the list and watch the standard output and the window for the details.

### ***Listing 12-10.*** Using `ChoiceBox` with a Preselected Item

```
// ChoiceBoxTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.beans.value.ObservableValue;
import javafx.scene.Scene;
import javafx.scene.control.ChoiceBox;
import javafx.scene.control.Label;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class ChoiceBoxTest extends Application {
    public static void main(String[] args) {
```

```

        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Label seasonLbl = new Label("Select a Season:");
        ChoiceBox<String> seasons = new ChoiceBox<>();
        seasons.getItems().addAll("Spring", "Summer",
        "Fall", "Winter");

        // Select the first season from the list
        seasons.getSelectionModel().selectFirst();

        // Add ChangeListeners to track change in selected
        index and item. Only
        // one listener is necessary if you want to track
        change in selection
        seasons.getSelectionModel().selectedItemProperty()
            .addListener(this::itemChanged)
    ;
        seasons.getSelectionModel().selectedIndexProperty()
            .addListener(this::indexChanged
    );

        Label selectionMsgLbl = new Label("Your
selection:");
        Label selectedValueLbl = new Label("None");

        // Bind the value property to the text property of
        the Label
        selectedValueLbl.textProperty().bind(seasons.valuePr
operty());

        // Display controls in a GridPane
        GridPane root = new GridPane();
        root.setVgap(10);
        root.setHgap(10);
        root.addRow(0, seasonLbl, seasons);
        root.addRow(1, selectionMsgLbl, selectedValueLbl);
        root.setStyle("-fx-padding: 10;" +
                    "-fx-border-style: solid inside;" +
                    "-fx-border-width: 2;" +
                    "-fx-border-insets: 5;" +
                    "-fx-border-radius: 5;" +
                    "-fx-border-color: blue;");

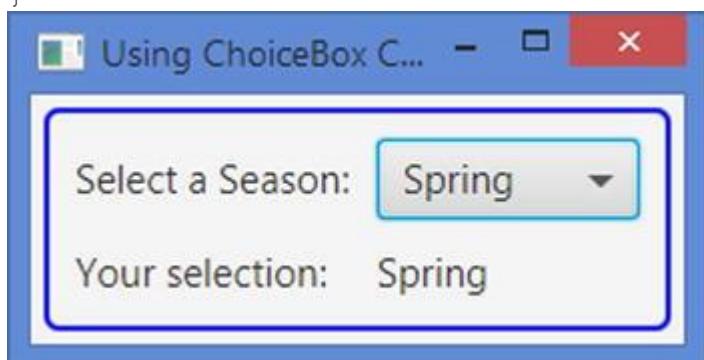
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using ChoiceBox Controls");
        stage.show();
    }

    // A change listener to track the change in selected item
    public void itemChanged(ObservableValue<? extends String>
observable,

```

```
        String oldValue,
        String newValue) {
    System.out.println("Itemchanged: old = " + oldValue
+ ",",
                      newValue);
}

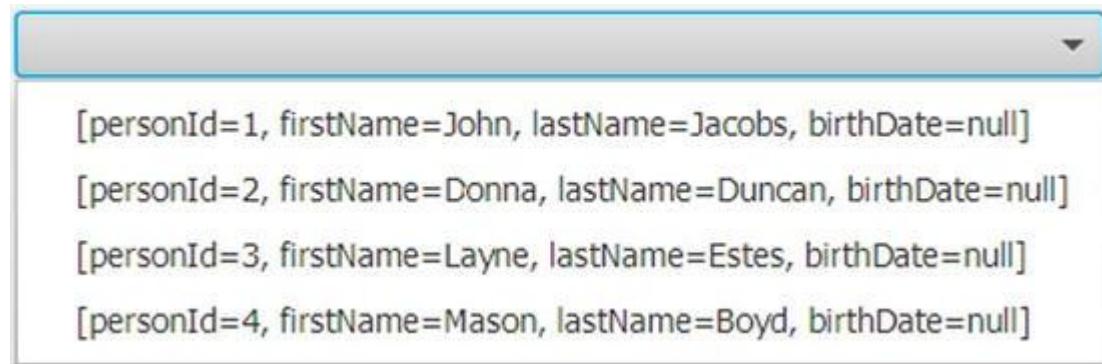
// A change listener to track the change in selected index
public void indexChanged(ObservableValue<? extends Number>
observable,
                          Number oldValue,
                          Number newValue) {
    System.out.println("Indexchanged: old = " + oldValue
+ ", new = " + newValue);
}
```



**Figure 12-14.** A choice box with a preselected item

## Using Domain Objects in *ChoiceBox*

In the previous example, you used `String` objects as items in the choice box. You can use any object type as items. `ChoiceBox` calls the `toString()` method of every item and displays the returned value in the pop-up list. The following snippet of code creates a choice box and adds four `Person` objects as its items. Figure 12-15 shows the choice box in the showing state. Notice the items are displayed using the `String` object returned from `toString()` method of the `Person` class.



**Figure 12-15.** A choice box showing four Person objects as its list of items

Typically, the `toString()` method of an object returns a String that represents the state of the object. It is not meant to provide a customized string representation of the object to be displayed in a choice box. The `ChoiceBox` class contains a `converter` property. It is an `ObjectProperty` of the `StringConverter<T>` type. A `StringConverter<T>` object acts as a converter from the object type `T` to a string and vice versa. The class is declared abstract, as in the following snippet of code:

```
public abstract class StringConverter<T> {
    public abstract String toString(T object);
    public abstract T fromString(String string);
}
```

The `toString(T object)` method converts the object of type `T` to a string. The `fromString(String string)` method converts a string to a `T` object.

By default, the `converter` property in a choice box is `null`. If it is set, the `toString(T object)` method of the converter is called to get the list of items instead of the `toString()` method of the class of the item. The `PersonStringConverter` class shown in Listing 12-11 can act as a converter in a choice box. Notice that you are treating the argument `string` in the `fromString()` method as the name of a person and trying to construct a `Person` object from it. You do not need to implement the `fromString()` method for a choice box. It will be used in a `ComboBox`, which I will discuss next. The `ChoiceBox` will use only the `toString(Person p)` method.

### **Listing 12-11.** A Person to String Converter

```
// PersonStringConverter.java
package com.jdojo.control;

import com.jdojo.mvc.model.Person;
```

```

import javafx.util.StringConverter;

public class PersonStringConverter extends
StringConverter<Person> {
    @Override
    public String toString(Person p) {
        return p == null? null : p.getLastName() + ", "
+ p.getFirstName();
    }

    @Override
    public Person fromString(String string) {
        Person p = null;
        if (string == null) {
            return p;
        }

        int commaIndex = string.indexOf(",");
        if (commaIndex == -1) {
            // Treat the string as first name
            p = new Person(string, null, null);
        } else {
            // Ignoring string bounds check for brevity
            String firstName = string.substring(commaIndex
+ 2);
            String lastName = string.substring(0,
commaIndex);
            p = new Person(firstName, lastName, null);
        }

        return p;
    }
}

```

The following snippet of code uses a converter in a `ChoiceBox` to convert `Person` objects in its list of items to strings. Figure 12-16 shows the choice box in the showing state.

```

import com.jdojo.mvc.model.Person;
import javafx.scene.control.ChoiceBox;
...
ChoiceBox<Person> persons = new ChoiceBox<>();

// Set a converter to convert a Person object to a String object
persons.setConverter(new PersonStringConverter());

// Add five person objects to the ChoiceBox
persons.getItems().addAll(new Person("John", "Jacobs", null),
    new Person("Donna", "Duncan", null),
    new Person("Layne", "Estes", null),
    new Person("Mason", "Boyd", null));

```



**Figure 12-16.** Person objects using a converter in a choice box

### Allowing Nulls in *ChoiceBox*

Sometimes a choice box may allow the user to select `null` as a valid choice. This can be achieved by using `null` as an item in the list of choices, as shown in the following code:

```
ChoiceBox<String> seasons = new ChoiceBox<>();
seasons.getItems().addAll(null, "Spring", "Summer", "Fall",
"Winter");
```

The above snippet of code produces a choice box as shown in Figure 12-17. Notice that the `null` item is shown as an empty space.



**Figure 12-17.** Null as a choice in a choice box

It is often required that the `null` choice be shown as a custom string, for example, "[None]". This can be accomplished using a converter. In the previous section, you used a converter to customize the choices

for Person objects. Here you will use the converter to customize the choice item for null. You can do both in one converter as well. The following snippet of code uses a converter with a ChoiceBox to convert a null choice as "[None]". Figure 12-18 shows the resulting choice box.

```
ChoiceBox<String> seasons = new ChoiceBox<>();
seasons.getItems().addAll(null, "Spring", "Summer", "Fall",
"Winter");

// Use a converter to convert null to "[None]"
seasons.setConverter(new StringConverter<String>() {
    @Override
    public String toString(String string) {
        return (string == null) ? "[None]" : string;
    }

    @Override
    public String fromString(String string) {
        return string;
    }
});
```



**Figure 12-18.** A null choice in a choice box converted as "[None]"

### Using Separators in ChoiceBox

Sometimes you may want to separate choices into separate groups. Suppose you want to show fruits and cooked items in a breakfast menu, and you want to separate one from the other. You would use an instance of the Separator class to achieve this. It appears as a horizontal line in the list of choices. A Separator is not selectable. The following snippet

of code creates a choice box with one of its items as a Separator. Figure 12-19 shows the choice box in the showing state.

```
ChoiceBox breakfasts = new ChoiceBox();
breakfasts.getItems().addAll("Apple", "Banana", "Strawberry",
                            new Separator(),
                            "Apple Pie", "Donut", "Hash Brown");
```



**Figure 12-19.** A choice box using a separator

### Styling a *ChoiceBox* with CSS

The default CSS style-class name for a *ChoiceBox* is `choice-box`. The *ChoiceBox* class supports a `showing` CSS pseudo-class, which applies when the `showing` property is true.

The *ChoiceBox* control contains two substructures: open-button and arrow. You can style them to change their appearance. Both are instances of *StackPane*. *ChoiceBox* shows the selected item in a *Label*. The list of choices are shown in a *ContextMenu* whose ID is set to `choice-box-popup-menu`. Each choice is displayed in a menu item whose IDs are set to `choice-box-menu-item`. The following styles customize the *ChoiceBox* control. Currently, there is no way to customize the pop-up menu for an individual choice box. The style will affect all instances of *ChoiceBox* control at the level (scene or layout pane) at which it is set.

```
/* Set the text color and font size for the selected item in the
control */
.choice-box .label {
```

```

        -fx-text-fill: blue;
        -fx-font-size: 8pt;
    }

/* Set the text color and text font size for choices in the popup
list */
#choice-box-menu-item * {
    -fx-text-fill: blue;
    -fx-font-size: 8pt;
}

/* Set background color of the arrow */
.choice-box .arrow {
    -fx-background-color: blue;
}

/* Set the background color for the open-button area */
.choice-box .open-button {
    -fx-background-color: yellow;
}

/* Change the background color of the popup */
#choice-box-popup-menu {
    -fx-background-color: yellow;
}

```

**Tip** There is a bug in applying the styles to the ChoiceBox pop-up. Styles are not effective until the pop-up is opened twice.

## Understanding the **ComboBox** Control

ComboBox is used to let a user select an item from a list of items. You can think of ComboBox as an advanced version of ChoiceBox. ComboBox is highly customizable. The ComboBox class inherits from ComboBoxBase class, which provides the common functionality for all ComboBox-like controls, such as ComboBox, ColorPicker, and DatePicker. If you want to create a custom control that will allow users to select an item from a pop-up list, you need to inherit your control from the ComboBoxBase class.

The items list in a ComboBox may comprise any type of objects. ComboBox is a parameterized class. The parameter type is the type of the items in the list. If you want to store mixed types of items in a ComboBox, you can use its raw type, as in the following code:

```

// Create a ComboBox for any type of items
ComboBox seasons = new ComboBox();

// Create a ComboBox for String items
ComboBox<String> seasons = new ComboBox<String>();

```

You can specify the list items while creating a ComboBox, as in the following code:

```
ObservableList<String> seasonList
= FXCollections.<String>observableArrayList(
    "Spring", "Summer", "Fall",
    "Winter");
ComboBox<String> seasons = new ComboBox<>(seasonList);
```

After you create a combo box, you can add items to its list of items using the `items` property, which is of the `ObjectProperty<ObservableList<T>>` type, in which `T` is the type parameter for the combo box, as in the following code:

```
ComboBox<String> seasons = new ComboBox<>();
seasons.getItems().addAll("Spring", "Summer", "Fall", "Winter");
```

Like `ChoiceBox`, `ComboBox` needs to track the selected item and its index in the list of items. It uses a separate object, called *selection model*, for this purpose. The `ComboBox` class contains a `selectionModel` property to store the item selection details. `ComboBox` uses an object of the `SingleSelectionModel` class as its selection model. The selection model lets you select an item from the list of items and lets you add `ChangeListeners` to track changes in index and item selections. Please refer to the section “Understanding the *ChoiceBox Control*” for more details on using a selection model.

Unlike `ChoiceBox`, `ComboBox` can be editable.

Its `editable` property specifies whether or not it is editable. By default, it is not editable. When it is editable, it uses a `TextField` control to show the selected or entered item. The `editor` property of the `ComboBox` class stores the reference of the `TextField` and it is `null` if the combo box is not editable, as shown in the following code:

```
ComboBox<String> breakfasts = new ComboBox<>();

// Add some items to choose from
breakfasts.getItems().addAll("Apple", "Banana", "Strawberry");

// By making the control editable, let users enter an item
breakfasts.setEditable(true);
```

`ComboBox` has a `value` property that stores the currently selected or entered value. Note that when a user enters a value in an editable combo box, the entered string is converted to the item type `T` of the combo box. If the item type is not a string, a `StringConverter<T>` is needed to convert the `String` value to type `T`. I will present an example of this shortly.

You can set a prompt text for a combo box that is displayed when the control is editable, it does not have focus, and its `value` property is `null`. The prompt text is stored in the `promptText` property, which is of the `StringProperty` type, as in the following code:

```
breakfasts.setPromptText("Select/Enter an item"); // Set a prompt text
```

The ComboBox class contains a placeholder property, which stores a Node reference. When the items list is empty or null, the placeholder node is shown in the pop-up area. The following snippet of code sets a Label as a placeholder:

```
Label placeHolder = new Label("List is empty.\nPlease enter an item");
breakfasts.setPlaceholder(placeHolder);
```

The program in Listing 12-12 creates

two ComboBox controls: seasons and breakfasts. The combo box having the list of seasons is not editable. The combo box having the list of breakfast items is editable. Figure 12-20 shows the screenshot when the user selected a season and entered a breakfast item, Donut, which is not in the list of breakfast items. A Label control displays the user selection. When you enter a new value in the breakfast combo box, you need to change the focus, press the Enter key, or open the pop-up list to refresh the message Label.

### ***Listing 12-12.*** Using ComboBox Controls

```
// ComboBoxTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;
import javafx.scene.Scene;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ComboBoxTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Label seasonsLbl = new Label("Season:");
        ComboBox<String> seasons = new ComboBox<>();
        seasons.getItems().addAll("Spring", "Summer",
        "Fall", "Winter");

        Label breakfastsLbl = new Label("Breakfast:");
        ComboBox<String> breakfasts = new ComboBox<>();
        breakfasts.getItems().addAll("Apple", "Banana",
        "Strawberry");
        breakfasts.setEditable(true);
    }
}
```

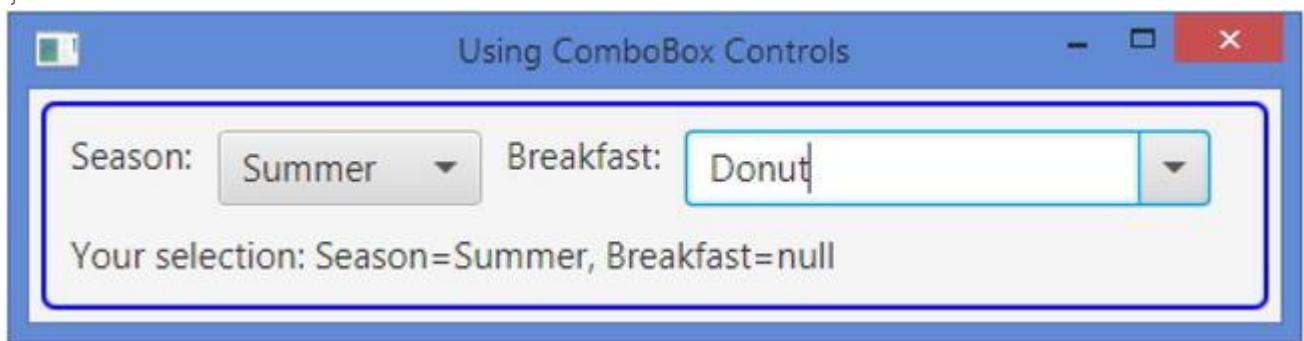
```

        // Show the user's selection in a Label
        Label selectionLbl = new Label();
        StringProperty str = new SimpleStringProperty("Your
selection: ");
        selectionLbl.textProperty().bind(str.concat("Season=")
                .concat(seasons.valuePro
perty()))
                .concat(", Breakfast=")
                .concat(breakfasts.value
Property()));

        HBox row1 = new HBox(seasonsLbl, seasons,
breakfastsLbl, breakfasts);
        row1.setSpacing(10);
        VBox root = new VBox(row1, selectionLbl);
        root.setSpacing(10);
        root.setStyle("-fx-padding: 10;" +
                  "-fx-border-style: solid inside;" +
                  "-fx-border-width: 2;" +
                  "-fx-border-insets: 5;" +
                  "-fx-border-radius: 5;" +
                  "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using ComboBox Controls");
        stage.show();
    }
}

```



**Figure 12-20.** Two ComboBox controls: one noneditable and one editable

### Detecting Value Change in ComboBox

Detecting an item change in a noneditable combo box is easily performed by adding a `ChangeListener` to the `selectedIndex` or `selectedItem` property of its selection model. Please refer to the “Understanding the *ChoiceBox* Control” section for more details.

You can still use a ChangeListener for the selectedItem property to detect when the value in an editable combo box changes by selecting from the items list or entering a new value. When you enter a new value, the selectedIndex property does not change because the entered value does not exist in the items list.

Sometimes you want to perform an action when the value in a combo box changes. You can do so by adding an ActionEvent handler, which is fired when the value changes by any means. You would do this by setting it programmatically, selecting from items list, or entering a new value, as in the following code:

```
ComboBox<String> list = new ComboBox<>();
list.setOnAction(e -> System.out.println("Value changed"));
```

### Using Domain Objects in Editable ComboBox

In an editable ComboBox<T> where T is something other than String, you must set the converterproperty to a valid StringConverter<T>. Its `toString(T object)` method is used to convert the item object to a string to show it in the pop-up list. Its `fromString(String s)` method is called to convert the entered string to an item object. The `value` property is updated with the item object converted from the entered string. If the entered string cannot be converted to an item object, the `value` property is not updated.

The program in Listing 12-13 shows how to use a StringConverter in a combo box, which uses domain objects in its items list. The ComboBox uses Person objects. The PersonStringConverter class, as shown in Listing 12-11, is used as the StringConverter. You can enter a name in the format LastName, FirstName or FirstName in the ComboBox and press the Enter key. The entered name will be converted to a Person object and shown in the Label. The program ignores the error checking in name formatting. For example, if you enter Kishori as the name, it displays null, Kishori in the Label. The program adds a ChangeListener to the selectedItem and selectedIndex properties of the selection model to track the selection change. Notice that when you enter a string in the ComboBox, a change in selectedIndex property is not reported. An ActionEvent handler for the ComboBox is used to keep the values in the combo box and the text in the Label in sync.

### **Listing 12-13.** Using a StringConverter in a ComboBox

```
// ComboBoxWithConverter.java
package com.jdojo.control;
```

```
import com.jdojo.mvc.model.Person;
import javafx.application.Application;
import javafx.beans.value.ObservableValue;
import javafx.scene.Scene;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class ComboBoxWithConverter extends Application {
    Label userSelectionMsgLbl = new Label("Your selection:");
    Label userSelectionDataLbl = new Label("");

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

    @Override
    public void start(Stage stage) {
        Label personLbl = new Label("Select/Enter Person:");
        ComboBox<Person> persons = new ComboBox<>();
        persons.setEditable(true);
        persons.setConverter(new PersonStringConverter());
        persons.getItems().addAll(new Person("John",
        "Jacobs", null),
                new Person("Donna", "Duncan",
        null),
                new Person("Layne", "Estes",
        null),
                new Person("Mason", "Boyd",
        null));

        // Add ChangeListeners to the selectedItem and
        selectedIndex
        // properties of the selection model
        persons.getSelectionModel().selectedItemProperty()
            .addListener(this::personChange
d);
        persons.getSelectionModel().selectedIndexProperty()
            .addListener(this::indexChanged
);

        // Update the message Label when the value changes
        persons.setOnAction(e -> valueChanged(persons));

        GridPane root = new GridPane();
        root.addRow(0, personLbl, persons);
        root.addRow(1, userSelectionMsgLbl,
        userSelectionDataLbl);
        root.setStyle("-fx-padding: 10;" +
            "-fx-border-style: solid inside;" +
            "-fx-border-width: 2;" +
            "-fx-border-insets: 5;" +
            "-fx-border-radius: 5;" +
```

```

"-fx-border-color: blue;");

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("Using StringConverter in ComboBox");
stage.show();
}

public void valueChanged(ComboBox<Person> list) {
    Person p = list.getValue();
    String name = p.getLastName() + ", "
+ p.getFirstName();
    userSelectionDataLbl.setText(name);
}

// A change listener to track the change in item selection
public void personChanged(ObservableValue<? extends Person>
observable,
                           Person oldValue,
                           Person newValue) {
    System.out.println("Itemchanged: old = " + oldValue
+
                           ", new = " + newValue);
}

// A change listener to track the change in index selection
public void indexChanged(ObservableValue<? extends Number>
observable,
                           Number oldValue,
                           Number newValue) {
    System.out.println("Indexchanged: old = " + oldValue
+ ", new = " + newValue);
}
}

```

## Customizing the Height of Pop-up List

By default, `ComboBox` shows only ten items in the pop-up list. If the number of items is more than ten, the pop-up list shows a scrollbar. If the number of items is less than ten, the height of the pop-up list is shortened to show only the available items.

The `visibleRowCount` property of the `ComboBox` controls how many rows are visible in the pop-up list, as in the following code:

```

ComboBox<String> states = new ComboBox<>();
...
// Show five rows in the popup list
states.setVisibleRowCount(5);

```

## Using Nodes as Items in `ComboBox`

A combo box has two areas:

- Button area to display the selected item
- Pop-up area to display the items list

Both areas use `ListCells` to display items. A `ListCell` is a `Cell`. A `Cell` is a Labeled control to display some form of content that may have text, a graphic, or both. The pop-up area is a `ListView` that contains an instance of `ListCell` for each item in the list. I will discuss `ListView` in the next section.

Elements in the items list of a combo box can be of any type, including `Node` type. It is not recommended to add instances of the `Node` class directly to the items list. When nodes are used as items, they are added as the graphic to the cells. Scene graphics need to follow the rule that a node cannot be displayed in two places at the same time. That is, a node must be inside one container at a time. When a node from the items list is selected, the node is removed from the pop-up `ListView` cell and added to the button area. When the pop-up is displayed again, the selected node is not shown in the list as it is already showing in the button area. To avoid this inconsistency in display, avoid using nodes directly as items in a combo box.

Figure 12-21 show three views of a combo box created using the following snippet of code. Notice that the code adds three instances of `HBox`, which is a node to the items list. The figure labeled #1 shows the pop-up list when it is opened for the first time, and you see all three items correctly. The figure labeled #2 shows after the second item is selected and you see the correct item in the button area. At this time, the second item in the list, an `HBox` with a rectangle, was removed from the cell in the `ListView` and added to the cell in the button area. The figure labeled #3 shows the pop-up list when it is open for the second time. At this time, the second item is missing from the list because it is already selected. This problem was discussed in the previous paragraph.

```
Label shapeLbl = new Label("Shape:");
ComboBox<HBox> shapes = new ComboBox<>();
shapes.getItems().addAll(new HBox(new Line(0, 10, 20, 10), new
Label("Line")),
    new HBox(new Rectangle(0, 0, 20, 20), new
Label("Rectangle")),
    new HBox(new Circle(20, 20, 10), new
Label("Circle")));
```



**Figure 12-21.** Three views of a combo box with nodes in the items list

You can fix the display issue that occurs when you use nodes as items. The solution is to add nonnode items in the list and supply a cell factory to create the desired node inside the cell factory. You need to make sure that the nonnode items will provide enough pieces of information to create the node you wanted to insert. The next section explains how to use a cell factory.

### Using a Cell Factory in *ComboBox*

The `ComboBox` class contains a `cellFactory` property, which is declared as follows:

```
public ObjectProperty<Callback<ListView<T>, ListCell<T>>>
cellFactory;
```

`Callback` is an interface in the `javafx.util` package. It has a `call()` method that takes an argument of type `P` and returns an object of type `R`, as in the following code:

```
public interface Callback<P, R> {
    public R call(P param);
}
```

The declaration of the `cellFactory` property states that it stores a `Callback` object whose `call()` method receives a `ListView<T>` and returns a `ListCell<T>`. Inside the `call()` method, you create an instance of the `ListCell<T>` class and override the `updateItem(T item, boolean empty)` method of the `Cell` class to populate the cell.

Let's use a cell factory to display nodes in the button area and the pop-up area of a combo box. Listing 12-14 will be our starting point. It declares a `StringShapeCell` class, which inherits from the `ListCell<String>` class. You need to update its content in its `updateItem()` method, which is automatically called. The method receives the item, which in this case is `String`, and a `boolean` argument indicating whether the cell is empty. Inside the method, you call the method in the superclass first. You derive a shape from the string argument and set the text and graphic in the cell. The shape is set as the graphic. The `getShape()` method returns a `Shape` from a `String`.

***Listing 12-14.*** A Custom `ListCell` that Displays a Shape and Its Name

```
// StringShapeCell.java
package com.jdojo.control;

import javafx.scene.control.ListCell;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Line;
import javafx.scene.shape.Rectangle;
import javafx.scene.shape.Shape;

public class StringShapeCell extends ListCell<String> {
    @Override
    public void updateItem(String item, boolean empty) {
        // Need to call the super first
        super.updateItem(item, empty);

        // Set the text and graphic for the cell
        if (empty) {
            setText(null);
            setGraphic(null);
        } else {
            setText(item);
            Shape shape = this.getShape(item);
            setGraphic(shape);
        }
    }

    public Shape getShape(String shapeType) {
        Shape shape = null;
        switch (shapeType.toLowerCase()) {
            case "line":
                shape = new Line(0, 10, 20, 10);
                break;
            case "rectangle":
                shape = new Rectangle(0, 0, 20, 20);
                break;
            case "circle":
                shape = new Circle(10, 10, 10);
                break;
        }
        return shape;
    }
}
```

```

        shape = new Circle(20, 20, 10);
        break;
    default:
        shape = null;
    }
    return shape;
}
}

```

The next step is to create a `Callback` class, as shown in Listing 12-15. The program in this listing is very simple. Its `call()` method returns an object of the `StringShapeCell` class. The class will act as a cell factory for `ComboBox`.

### ***Listing 12-15.***

A `Callback` Implementation for `Callback<ListView<String>, ListCell<String>>`

```

// ShapeCellFactory.java
package com.jdojo.control;

import javafx.scene.control.ListCell;
import javafx.scene.control.ListView;
import javafx.util.Callback;

public class ShapeCellFactory implements
Callback<ListView<String>, ListCell<String>> {
    @Override
    public ListCell<String> call(ListView<String> listview) {
        return new StringShapeCell();
    }
}

```

The program in Listing 12-16 shows how to use a custom cell factory and button cell in a combo box. The program is very simple. It creates a combo box with three `String` items. It sets an object of the `ShapeCellFactory` as the cell factory, as in the following code:

```
// Set the cellFactory property
shapes.setCellFactory(new ShapeCellFactory());
```

Setting the cell factory is not enough in this case. It will only resolve the issue of displaying the shapes in the pop-up area. When you select a shape, it will display the `String` item, not the shape, in the button area. To make sure, you see the same item in the list for selection and after you select one, you need to set the `buttonCell` property, as in the following code:

```
// Set the buttonCell property
shapes.setButtonCell(new StringShapeCell());
```

Notice the use of the `StringShapeCell` class in the `buttonCell` property and `ShapeCellFactory` class.

Run the program in Listing 12-16. You should be able to select a shape from the list and the shape should be displayed in the combo box correctly. Figure 12-22 shows three views of the combo box.

### ***Listing 12-16.*** Using a Cell Factory in a Combo Box

```
// ComboBoxCellFactory.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class ComboBoxCellFactory extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

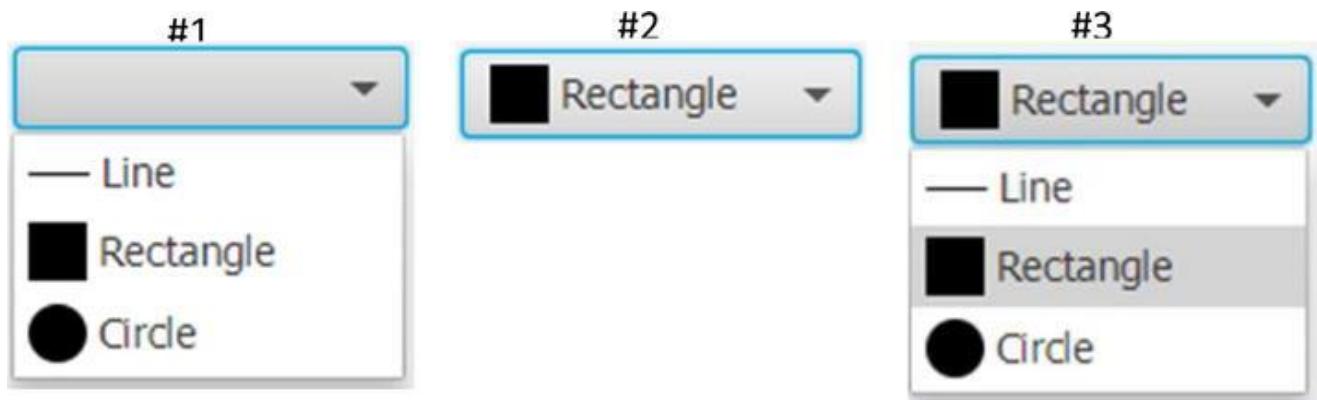
    @Override
    public void start(Stage stage) {
        Label shapeLbl = new Label("Shape:");
        ComboBox<String> shapes = new ComboBox<>();
        shapes.getItems().addAll("Line", "Rectangle",
        "Circle");

        // Set the cellFactory property
        shapes.setCellFactory(new ShapeCellFactory());

        // Set the buttonCell property
        shapes.setButtonCell(new StringShapeCell());

        HBox root = new HBox(shapeLbl, shapes);
        root.setStyle("-fx-padding: 10;" +
                    "-fx-border-style: solid inside;" +
                    "-fx-border-width: 2;" +
                    "-fx-border-insets: 5;" +
                    "-fx-border-radius: 5;" +
                    "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using CellFactory in ComboBox");
        stage.show();
    }
}
```



**Figure 12-22.** Three views of a combo box with a cell factory

Using a custom cell factory and button cell in a combo box gives you immense power to customize the look of the pop-up list and the selected item. If using a cell factory looks hard or confusing to you, keep in mind that a cell is a `Labeled` control and you are setting the text and graphic in that `Labeledcontrol` inside the `updateItem()` method.

The `Callback` interface comes into play because the `ComboBox` control needs to give you a chance to create a cell when it needs it. Otherwise, you would have to know how many cells to create and when to create them. There is nothing more to it.

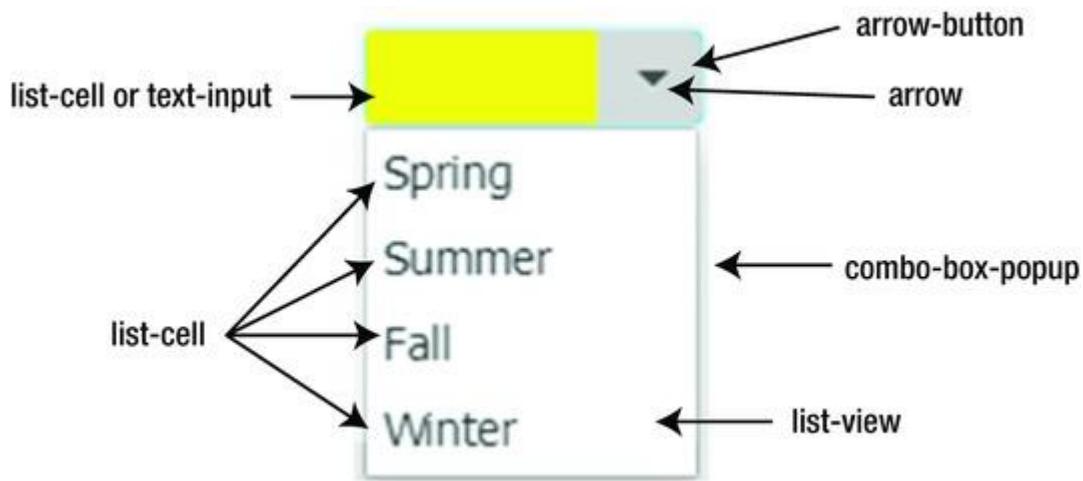
The `ComboBoxBase` class provides four properties that can also be used with `ComboBox`:

- `onShowing`
- `onShown`
- `onHiding`
- `onHidden`

These properties are of the type `ObjectProperty<EventHandler<Event>>`. You can set an event handler to these properties, which will be called before the pop-up list is shown, after it is shown, before it is hidden, and after it is hidden. For example, the `onShowing` event handlers are handy when you want to customize the pop-up list just before it is shown.

### Styling `ComboBox` with CSS

The default CSS style-class name for a `ComboBox` is `combo-box`. A combo box contains several CSS substructures, as shown in Figure 12-23.



**Figure 12-23.** Substructures of a combo box that can be styled separately using CSS

The CSS names for the substructure are:

- arrow-button
- list-cell
- text-input
- combo-box-popup

An arrow-button contains a substructure called arrow. Both arrow-button and arrow are instances of StackPane. The list-cell area represents the ListCell used to show the selected item in a noneditable combo box. The text-input area is the TextField used to show the selected or entered item in an editable combo box. The combo-box-popup is the Popup control that shows the pop-up list when the button is clicked. It has two substructures: list-view and list-cell. The list-view is the ListView control that shows the list of items, and list-cell represents each cell in the ListView. The following CSS styles customize the appearance of some substructures of ComboBox:

```
/* The ListCell that shows the selected item in a non-editable
ComboBox */
.combo-box .list-cell {
    -fx-background-color: yellow;
}

/* The TextField that shows the selected item in an editable
ComboBox */
.combo-box .text-input {
    -fx-background-color: yellow;
}

/* Style the arrow button area */
```

```
.combo-box .arrow-button {
    -fx-background-color: lightgray;
}

/* Set the text color in the popup list for ComboBox to blue */
.combo-box-popup .list-view .list-cell {
    -fx-text-fill: blue;
}
```

## Understanding the *ListView* Control

`ListView` is used to allow a user to select one item or multiple items from a list of items. Each item in `ListView` is represented by an instance of the `ListCell` class, which can be customized. The items list in a `ListView` may contain any type of objects. `ListView` is a parameterized class. The parameter type is the type of the items in the list. If you want to store mixed types of items in a `ListView`, you can use its raw type, as shown in the following code:

```
// Create a ListView for any type of items
ListView seasons = new ListView();

// Create a ListView for String items
ListView<String> seasons = new ListView<String>();
```

You can specify the list items while creating a `ListView`, as in the following code:

```
ObservableList<String> seasonList
= FXCollections.<String>observableArrayList(
    "Spring", "Summer", "Fall",
    "Winter");
ListView<String> seasons = new ListView<>(seasonList);
```

After you create a `ListView`, you can add items to its list of items using the `items` property, which is of

the `ObjectProperty<ObservableList<T>>` type in which `T` is the type parameter for the `ListView`, as in the following code:

```
ListView<String> seasons = new ListView<>();
seasons.getItems().addAll("Spring", "Summer", "Fall", "Winter");
ListView sets its preferred width and height, which are normally not the width and height that you want for your control. It would have helped developers if the control had provided a property such as visibleItemCount. Unfortunately, the ListView API does not support such a property. You need to set them to reasonable values in your code, as follows:
```

```
// Set preferred width = 100px and height = 120px
seasons.setPrefSize(100, 120);
```

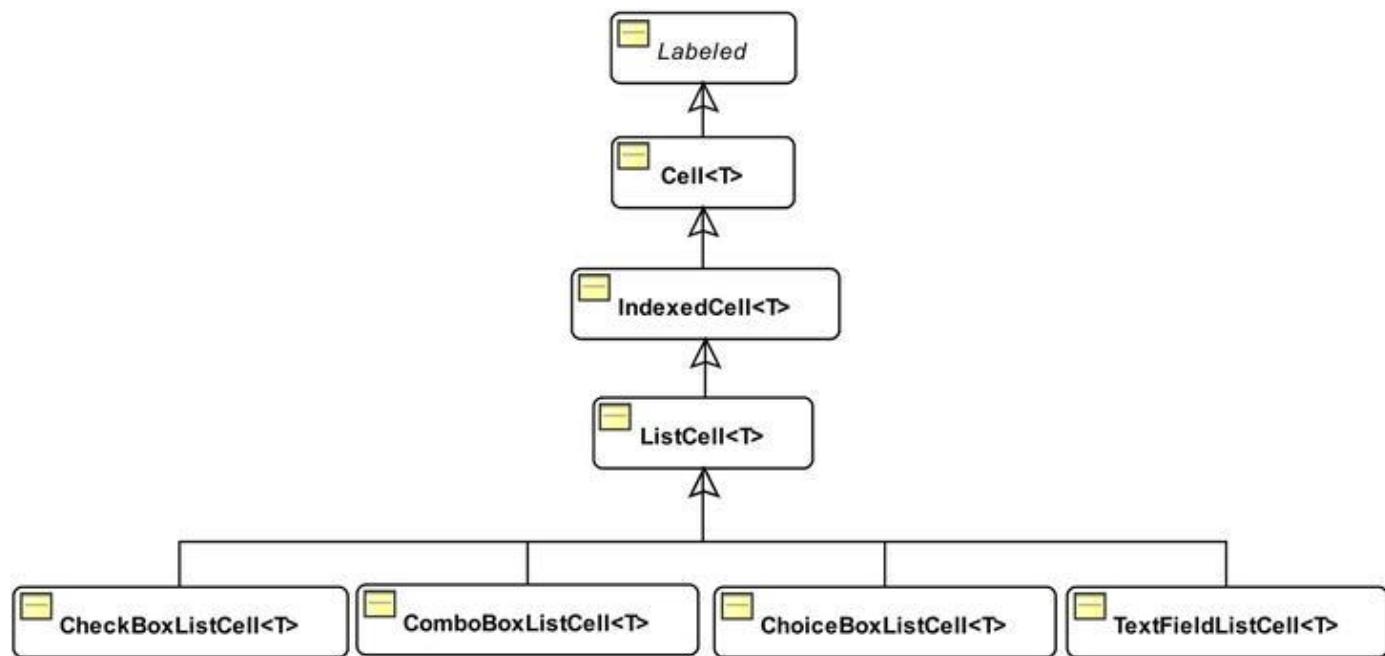
If the space needed to display items is larger than what is available, a vertical, a horizontal, or both scrollbars are automatically added.

The `ListView` class contains a `placeholder` property, which stores a `Node` reference. When the items list is empty or null, the placeholder node is shown in the list area of the `ListView`. The following snippet of code sets a `Label` as a placeholder:

```
Label placeHolder = new Label("No seasons available for selection.");
seasons.setPlaceholder(placeHolder);
```

`ListView` offers a scrolling feature. Use the `scrollTo(int index)` or `scrollTo(T item)` method to scroll to a specified `index` or `item` in the list. The specified `index` or `item` is made visible, if it is not already visible. The `ListView` class fires a `ScrollToEvent` when scrolling takes place using the `scrollTo()` method or by the user. You can set an event handler using the `setOnScrollTo()` method to handle scrolling.

Each item in a `ListView` is displayed using an instance of the `ListCell` class. In essence, a `ListCell` is a labeled control that is capable of displaying text and a graphic. Several subclasses of `ListCell` exist to give `ListView` items a custom look. `ListView` lets you specify a `Callback` object as a *cell factory*, which can create custom list cells. A `ListView` does not need to create as many `ListCell` objects as the number items. It can have only as many `ListCell` object as the number of visible items on the screen. As items are scrolled, it can reuse the `ListCell` objects to display different items. Figure 12-24 shows a class diagram for `ListCell`-related classes.



**Figure 12-24.** A class diagram for `ListCell`-related classes

Cells are used as building blocks in different types of controls. For example, ListView, TreeView, and TableView controls use cells in one form or another to display and edit their data. The Cell class is the superclass for all cells. You can override its `updateItem(T object, boolean empty)` and take full control of how the cell is populated. This method is called automatically by these controls when the item in the cell needs to be updated. The Cell class declares several useful properties: `editable`, `editing`, `empty`, `item`, and `selected`. When a Cell is empty, which means it is not associated with any data item, its `empty` property is true.

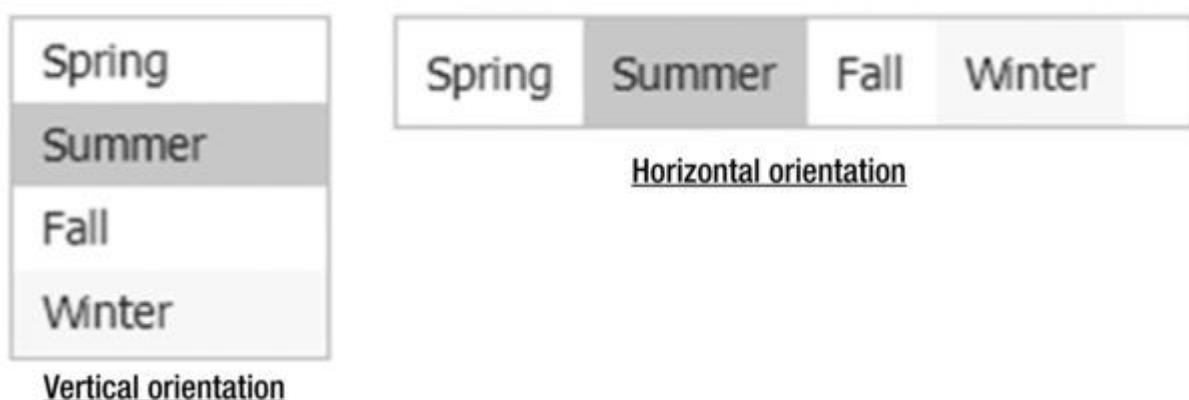
The IndexedCell class adds an `index` property, which is the index of the item in the underlying model. Suppose a ListView uses an ObservableList as a model. The list cell for the second item in the ObservableList will have index 1 (index starts at 0). The cell index facilitates customization of cells based on their indices, for example, using different colors for cells at odd and even index cells. When a cell is empty, its index is -1.

### Orientation of a ListView

The items in a ListView may be arranged vertically in a single column (default) or horizontally in a single row. It is controlled by the `orientation` property, as shown in the following code:

```
// Arrange list of seasons horizontally
seasons.setOrientation(Orientation.HORIZONTAL);
```

Figure 12-25 shows two instances of ListView: one uses vertical orientation and one horizontal orientation. Notice that the odd and even rows or columns have different background colors. This is the default look of the ListView. You can change the appearance using a CSS. Please refer to the “Styling ListView with CSS” section for details.



**Figure 12-25.** Two instances of ListView having the same items but different orientations

## Selection Model in *ListView*

*ListView* has a selection model that stores the selected state of its items. Its `selectionModel` property stores the reference of the selection model. By default, it uses an instance of the `MultipleSelectionModel` class. You can use a custom selection model, however, that is rarely needed. The selection model can be configured to work in two modes:

- Single selection mode
- Multiple selection mode

In single selection mode, only one item can be selected at a time. If an item is selected, the previously selected item is deselected. By default, a *ListView* supports single selection mode. An item can be selected using a mouse or a keyboard. You can select an item using a mouse-click. Using a keyboard to select an item requires that the *ListView* has focus. You can use the up/down arrow in a vertical *ListView* and the left/right arrow in a horizontal *ListView* to select items.

In multiple selection mode, multiple items can be selected at a time. Using only a mouse lets you select only one item at a time. Clicking an item selects the item. Clicking an item with the Shift key pressed selects all contiguous items. Clicking an item with the Ctrl key pressed selects a deselected item and deselects a selected item. You can use the up/down or left/right arrow key to navigate and the Ctrl key with the spacebar or Shift key with the spacebar to select multiple items. If you want a *ListView* to operate in multiple selection mode, you need to set the `selectionMode` property of its selection model, as in the following code:

```
// Use multiple selection mode
seasons.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);

// Set it back to single selection mode, which is the default for
// a ListView
seasons.getSelectionModel().setSelectionMode(SelectionMode.SINGLE);
```

The `MultipleSelectionModel` class inherits from the `SelectionModel` class, which contains `selectedIndex` and `selectedItem` properties.

The `selectedIndex` property is -1 if there is no selection. In single selection mode, it is the index of the currently selected item. In multiple selection mode, it is the index of the last selected item. In multiple selection mode, use the `getSelectedIndices()` method that returns

a read-only `ObservableList<Integer>` containing the indices of all selected items. If you are interested in listening for selection change in a `ListView`, you can add a `ChangeListener` to the `selectedIndex` property or a `ListChangeListener` to the `ObservableList` returned by the `getSelectedIndices()` method.

The `selectedItem` property is `null` if there is no selection. In single selection mode, it is the currently selected item. In multiple selection mode, it is the last selected item. In multiple selection mode, use the `getSelectedItems()` method that returns a read-only `ObservableList<T>` containing all selected items. If you are interested in listening for selection change in a `ListView`, you can add a `ChangeListener` to the `selectedItem` property or a `ListChangeListener` to the `ObservableList<T>` returned by the `getSelectedItems()` method.

The selection model of `ListView` contains several methods to select items in different ways:

- The `selectAll()` method selects all items.
- The `selectFirst()` and `selectLast()` methods select the first item and the last item, respectively.
- The `selectIndices(int index, int... indices)` method selects items at the specified indices. Indices outside the valid range are ignored.
- The `selectRange(int start, int end)` method selects all indices from the `start` index (inclusive) to the `end` index (exclusive).
- The `clearSelection()` and `clearSelection(int index)` methods clear all selection and the selection at the specified `index`, respectively.

The program in Listing 12-17 demonstrates how to use the selection model of a `ListView` for making selections and listening for selection change events. Figure 12-26 shows the window that results from running this code. Run the application and use a mouse or buttons on the window to select items in the `ListView`. The selection details are displayed at the bottom.

### ***Listing 12-17.*** Using `ListView` Selection Model

```
// ListViewSelectionModel.java
package com.jdojo.control;
```

```
import javafx.application.Application;
import javafx.beans.value.ObservableValue;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.ListView;
import javafx.scene.control.SelectionMode;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ListViewSelectionModel extends Application {
    private ListView<String> seasons;
    private final Label selectedItemsLbl = new Label("[None]");
    private final Label lastSelectedItemLbl = new
Label("[None]");

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

    @Override
    public void start(Stage stage) {
        Label seasonsLbl = new Label("Select Seasons:");
        seasons = new ListView<>();
        seasons.setPrefSize(120, 120);
        seasons.getItems().addAll("Spring", "Summer",
"Fall", "Winter");

        // Enable multiple selection
        seasons.getSelectionModel().setSelectionMode(Selecti
onMode.MULTIPLE);

        // Add a selection change listener
        seasons.getSelectionModel()
            .selectedItemProperty()
            .addListener(this::selectionChanged);

        // Add some buttons to assist in selection
        Button selectAllBtn = new Button("Select All");
        selectAllBtn.setOnAction(e ->
seasons.getSelectionModel().selectAll());

        Button clearAllBtn = new Button("Clear All");
        clearAllBtn.setOnAction(
            e ->
seasons.getSelectionModel().clearSelection());

        Button selectFirstBtn = new Button("Select First");
        selectFirstBtn.setOnAction(
            e ->
seasons.getSelectionModel().selectFirst());
    }
}
```

```
Button selectLastBtn = new Button("Select Last");
selectLastBtn.setOnAction(e ->
seasons.getSelectionModel().selectLast());

Button selectNextBtn = new Button("Select Next");
selectNextBtn.setOnAction(e ->
seasons.getSelectionModel().selectNext());

Button selectPreviousBtn = new Button("Select
Previous");
selectPreviousBtn.setOnAction(
e ->
seasons.getSelectionModel().selectPrevious());

// Let all buttons expand as needed
selectAllBtn.setMaxWidth(Double.MAX_VALUE);
clearAllBtn.setMaxWidth(Double.MAX_VALUE);
selectFirstBtn.setMaxWidth(Double.MAX_VALUE);
selectLastBtn.setMaxWidth(Double.MAX_VALUE);
selectNextBtn.setMaxWidth(Double.MAX_VALUE);
selectPreviousBtn.setMaxWidth(Double.MAX_VALUE);

// Display controls in a GridPane
GridPane root = new GridPane();
root.setHgap(10);
root.setVgap(5);

// Add buttons to two VBox objects
VBox singleSelectionBtns = new VBox(selectFirstBtn,
selectNextBtn,
selectLastBtn,
selectPreviousBtn);

VBox allSelectionBtns = new VBox(selectAllBtn,
clearAllBtn);
root.addColumn(0, seasonsLbl, seasons);
root.add(singleSelectionBtns, 1, 1, 1, 1);
root.add(allSelectionBtns, 2, 1, 1, 1);

// Add controls to display the user selection
Label selectionLbl = new Label("Your selection:");
root.add(selectionLbl, 0, 2);
root.add(selectedItemsLbl, 1, 2, 2, 1);

Label lastSelectionLbl = new Label("Last
selection:");
root.add(lastSelectionLbl, 0, 3);
root.add(lastSelectedItemLbl, 1, 3, 2, 1);

root.setStyle("-fx-padding: 10;" +
"-fx-border-style: solid inside;" +
"-fx-border-width: 2;" +
"-fx-border-insets: 5;" +
"-fx-border-radius: 5;" +
"-fx-border-color: blue;");
```

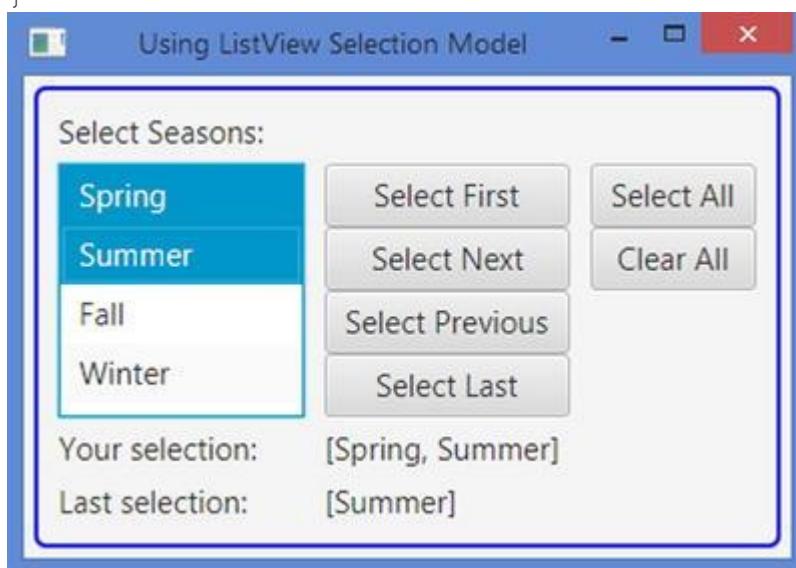
```

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using ListView Selection Model");
        stage.show();
    }

    // A change listener to track the change in item selection
    public void selectionChanged(ObservableValue<? extends
String> observable,
                                String oldValue,
                                String newValue) {
        String lastItem = (newValue == null)? "[None]": "[" +
+ newValue + "]";
        lastSelectedItemLbl.setText(lastItem);

        ObservableList<String> selectedItems =
            seasons.getSelectionModel().getSel
ectedItems();
        String selectedValues =
            (selectedItems.isEmpty())?"[None]":selectedIt
ems.toString();
        this.selectedItemsLbl.setText(selectedValues);
    }
}

```



**Figure 12-26.** A ListView with several buttons to make selections

### Using Cell Factory in ListView

Each item in a ListView is displayed in an instance of ListCell, which is a Labeled control. Recall that a Labeled control contains text and a graphic. The ListView class contains a cellFactory property that lets you use custom cells for its items. The property type is ObjectProperty<Callback<ListView<T>, ListCell<T>>>.

The reference of the `ListView` is passed to the `call()` method of the `Callback` object and it returns an instance of the `ListCell` class. In a large `ListView`, say 1,000 items, the `ListCell` returned from the cell factory may be reused. The control needs to create only the number of cells that are visible. Upon scrolling, it may reuse the cells that went out of the view to display newly visible items.

The `updateItem()` method of the `ListCell` receives the reference of the new item.

By default, a `ListView` calls the `toString()` method of its items and it displays the string in its cell. In the `updateItem()` method of your custom `ListCell`, you can populate the text and graphic for the cell to display anything you want in the cell based on the item in that cell.

**Tip** You used a custom cell factory for the pop-up list of the combo box in the previous section. The pop-up list in a combo box uses a `ListView`. Therefore, using a custom cell factory in a `ListView` would be the same as discussed in the earlier combo box section.

The program in Listing 12-18 shows how to use a custom cell factory to display the formatted names of `Person` items. Figure 12-27 shows the resulting window after running the code. The snippet of code in the program creates and sets a custom cell factory.

The `updateItem()` method of the `ListCell` formats the name of the `Person` object and adds a serial number that is the index of the cell plus one.

### ***Listing 12-18.*** Using a Custom Cell Factory for `ListView`

```
// ListViewDomainObjects.java
package com.jdojo.control;

import com.jdojo.mvc.model.Person;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.ListCell;
import javafx.scene.control.ListView;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;
import javafx.util.Callback;

public class ListViewDomainObjects extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
```

```

        ListView<Person> persons = new ListView<>();
        persons.setPrefSize(150, 120);
        persons.getItems().addAll(new Person("John",
        "Jacobs", null),
                                new Person("Donna", "Duncan",
        null),
                                new Person("Layne", "Estes",
        null),
                                new Person("Mason", "Boyd",
        null));

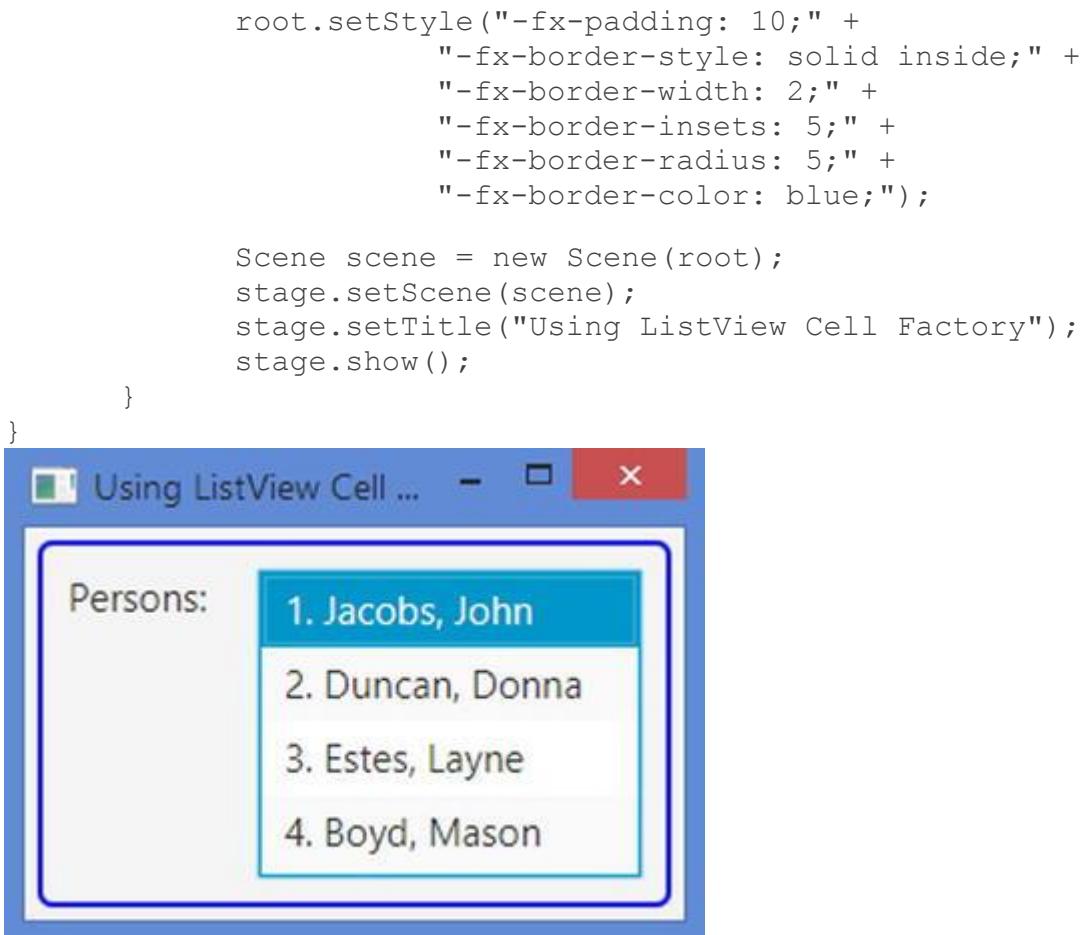
        // Add a custom cell factory to display formatted
        names of persons
        persons.setCellFactory(
            new
            Callback<ListView<Person>, ListCell<Person>>() {
                @Override
                public ListCell<Person> call(ListView<Person>
listView) {
                    return new ListCell<Person>() {
                        @Override
                        public void updateItem(Person
item, boolean empty) {
                            // Must call super
                            super.updateItem(item,
empty);

                            int index
                            String name = null;

                            // Format name
                            if (item == null || empty)
{
                                // No action to
                                perform
                            } else {
                                name = (index + 1)
+ ". " +
                            item.getLastName
                            () + ", " +
                            item.getFirstName
                            e();
                        }
                        this.setText(name);
                        setGraphic(null);
                    }
                };
            } );
    }

    HBox root = new HBox(new Label("Persons:"),
persons);
    root.setSpacing(20);
}

```



**Figure 12-27.** A `ListView` displaying `Person` objects in its list of items using a custom cell factory

### Using Editable `ListView`

The `ListView` control offers many customizations, and one of them is its ability to let users edit the items. You need to set two properties for a `Listview` before it can be edited:

- Set the `editable` property of the `ListView` to true.
- Set the `cellFactory` property of the `ListView` to a cell factory that produces an editable `ListCell`.

Select a cell and click to start editing. Alternatively, press the spacebar when a cell has focus to start editing. If a `ListView` is editable and has an editable cell, you can also use the `edit (int index)` method of the `ListView` to edit the item in the cell at the specified `index`.

**Tip** The `ListView` class contains a read-only `editingIndex` property. Its value is the index of the item being edited. Its value is -1 if no item is being edited.

JavaFX provides cell factories that let you edit a `ListCell` using `TextField`, `ChoiceBox`, `ComboBox`, and `CheckBox`. You can create a custom cell factory to edit cells in some other way. Instances of the `TextFieldListCell`, `ChoiceBoxListCell`, `ComboBoxListCell`, and `CheckBoxListCell` classes, as list cells in a `ListView`, provide editing support. These classes are included in the `javafx.scene.control.cell` package.

## Using a `TextField` to Edit `ListView` Items

An instance of the `TextFieldListCell` is a `ListCell` that displays an item in a `Label` when the item is not being edited and in a `TextField` when the item is being edited. If you want to edit a domain object to a `ListView`, you will need to use a `StringConverter` to facilitate the two-way conversion.

The `forListView()` static method of the `TextFieldListCell` class returns a cell factory configured to be used with `String` items. The following snippet of code shows how to set a `TextField` as the cell editor for a `ListView`:

```
ListView<String> breakfasts = new ListView<>();
...
breakfasts.setEditable(true);

// Set a TextField as the editor
Callback<ListView<String>, ListCell<String>> cellFactory =
    TextFieldListCell.forListView();
breakfasts.setCellFactory(cellFactory);
```

The following snippet of code shows how to set a `TextField` as the cell editor with a converter for a `ListView` that contains `Person` objects. The converter used in the code was shown in Listing 12-11. The converter object will be used to convert a `Person` object to a `String` for displaying and a `String` to a `Person` object after editing.

```
ListView<Person> persons = new ListView<>();
...
persons.setEditable(true);

// Set a TextField as the editor.
// Need to use a StringConverter for Person objects.
StringConverter<Person> converter = new PersonStringConverter();
Callback<ListView<Person>, ListCell<Person>> cellFactory
    = TextFieldListCell.forListView(converter);
persons.setCellFactory(cellFactory);
```

The program in Listing 12-19 shows how to edit a `ListView` item in a `TextField`. It uses a `ListView` of domain objects (`Person`) and a `ListView` of `String` objects. After running the program, double-click

on any items in the two `ListView`s to start editing. When you are done editing, press the Enter key to commit the changes.

### ***Listing 12-19.*** Using an Editable ListView

```
// ListViewEditing.java
package com.jdojo.control;

import com.jdojo.mvc.model.Person;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.ListCell;
import javafx.scene.control.ListView;
import javafx.scene.control.cell.TextFieldListCell;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;
import javafx.util.Callback;
import javafx.util.StringConverter;

public class ListViewEditing extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        ListView<String> breakfasts
= getBreakfastListView();
        ListView<Person> persons = getPersonListView();

        GridPane root = new GridPane();
        root.setHgap(20);
        root.setVgap(10);
        root.addRow(0, new Label("Double click an item to edit."),
0, 0, 2, 1);
        root.addRow(1, new Label("Persons:"), new
Label("Breakfasts:"));

        root.addRow(2, persons, breakfasts);
        root.setStyle("-fx-padding: 10;" +
"-fx-border-style: solid inside;" +
"-fx-border-width: 2;" +
"-fx-border-insets: 5;" +
"-fx-border-radius: 5;" +
"-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using ListView Cell Factory");
        stage.show();
    }

    public ListView<Person> getPersonListView() {
```

```

        ListView<Person> persons = new ListView<>();
        persons.setPrefSize(200, 120);
        persons.setEditable(true);
        persons.getItems().addAll(new Person("John",
"Jacobs", null),
                                new Person("Donna", "Duncan",
null),
                                new Person("Layne", "Estes",
null),
                                new Person("Mason", "Boyd",
null));

        // Set a TextField cell factory to edit the Person
        items. Also use a
        // StringConverter to convert a String to a Person
        and vice-versa
        StringConverter<Person> converter = new
        PersonStringConverter();
        Callback<ListView<Person>, ListCell<Person>>
        cellFactory =
                TextFieldListCell.forListView(converter);
        persons.setCellFactory(cellFactory);

        return persons;
    }

    public ListView<String> getBreakfastListView() {
        ListView<String> breakfasts = new ListView<>();
        breakfasts.setPrefSize(200, 120);
        breakfasts.setEditable(true);
        breakfasts.getItems().addAll("Apple", "Banana",
"Donut", "Hash Brown");

        // Set a TextField cell factory to edit the String
        items
        Callback<ListView<String>, ListCell<String>>
        cellFactory =
                TextFieldListCell.forListView();
        breakfasts.setCellFactory(cellFactory);

        return breakfasts;
    }
}

```

## Using a **ChoiceBox/ComboBox** to Edit **ListView** Items

An instance of the `ChoiceBoxListCell` is a `ListCell` that displays an item in a `Label` when the item is not being edited and in a `ChoiceBox` when the item is being edited. If you want to edit a domain object to a `ListView`, you will need to use a `StringConverter` to facilitate two-way conversion. You need to supply the list of items to show in the choice box. Use

the `forListView()` static method of the `ChoiceBoxListCell` class to create a cell factory. The following snippet of code shows how to set a choice box as the cell editor for a `ListView`:

```
ListView<String> breakfasts = new ListView<>();
...
breakfasts.setEditable(true);

// Set a cell factory to use a ChoiceBox for editing
ObservableList<String> items =
    FXCollections.<String>observableArrayList("Apple",
"Banana", "Donut", "Hash Brown");
breakfasts.setCellFactory(ChoiceBoxListCell.forListView(items));
```

The program in Listing 12-20 uses a choice box to edit items in a `ListView`. Double-click an item in a cell to start editing. In edit mode, the cell becomes a choice box. Click the arrow to show the list of items to select. Using a combo box for editing is similar to using a choice box.

### ***Listing 12-20.*** Using a ChoiceBox for Editing Items in a ListView

```
// ListViewChoiceBoxEditing.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.ListView;
import javafx.scene.control.SelectionMode;
import javafx.scene.control.cell.ChoiceBoxListCell;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ListViewChoiceBoxEditing extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        ListView<String> breakfasts = new ListView<>();
        breakfasts.setPrefSize(200, 120);
        breakfasts.setEditable(true);
        breakfasts.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);

        // Let the user select a maximum of four breakfast
        items
            breakfasts.getItems().addAll("[Double click to
select]",
                "[Double click to select]",
                "[Double click to select]",
```

```

        " [Double click to select]");

        // The breakfast items to select from
        ObservableList<String> items
= FXCollections.<String>observableArrayList(
                "Apple", "Banana", "Donut", "Hash
Brown");

        // Set a ChoiceBox cell factory for editing
        breakfasts.setCellFactory(ChoiceBoxListCell.forListV
iew(items));

        VBox root = new VBox(new Label("Double click an item
to select."),
                new Label("Breakfasts:"),
                breakfasts);
        root.setSpacing(10);
        root.setStyle("-fx-padding: 10;" +
                "-fx-border-style: solid inside;" +
                "-fx-border-width: 2;" +
                "-fx-border-insets: 5;" +
                "-fx-border-radius: 5;" +
                "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using ListView Cell Factory");
        stage.show();
    }
}

```

## Using a Check Box to Edit *ListView* Items

The `CheckBoxListCell` class provides the ability to edit a `ListCell` using a check box. It draws a check box in the cell, which can be selected or deselected. Note that the third state, the *indeterminate* state, of the check box is not available for selection while using a check box to edit `ListView` items.

Using a check box to edit `ListView` items is a little different. You need to provide the `CheckBoxListCell` class with an `ObservableValue<Boolean>` object for each item in the `ListView`. Internally, the observable value is bound bidirectionally to the selected state of the check box. When the user selects or deselects an item in the `ListView` using the check box, the corresponding `ObservableValue` object is updated with a true or false value. If you want to know which item is selected, you will need to keep the reference of the `ObservableValue` object.

Let's redo our earlier breakfast example using a check box. The following snippet of code creates a map and adds all items as a key and a

corresponding `ObservableValue` item with false value. Using a false value, you want to indicate that the items will be initially deselected:

```
Map<String, ObservableValue<Boolean>> map = new HashMap<>();
map.put("Apple", new SimpleBooleanProperty(false));
map.put("Banana", new SimpleBooleanProperty(false));
map.put("Donut", new SimpleBooleanProperty(false));
map.put("Hash Brown", new SimpleBooleanProperty(false));
```

Now, you create an editable `ListView` with all keys in the map as its items:

```
ListView<String> breakfasts = new ListView<>();
breakfasts.setEditable(true);

// Add all keys from the map as items to the ListView
breakfasts.getItems().addAll(map.keySet());
```

The following snippet of code creates a `Callback` object.

Its `call()` method returns the `ObservableValue` object for the specified item passed to the `call()` method.

The `CheckBoxListCell` class will call the `call()` method of this object automatically:

```
Callback<String, ObservableValue<Boolean>> itemToBoolean
= (String item) -> map.get(item);
```

Now it is time to create and set a cell factory for the `ListView`.

The `forListView()` static method of the `CheckBoxListCell` class takes a `Callback` object as an argument. If your `ListView` contains domain objects, you can also provide a `StringConverter` to this method, using the following code:

```
// Set the cell factory
breakfasts.setCellFactory(CheckBoxListCell.forListView(itemToBoolean));
```

When the user selects or deselects an item using the check box, the corresponding `ObservableValue` in the map will be updated. To know whether an item in the `ListView` is selected, you need to look at the value in the `ObservableValue` object for that item.

The program in Listing 12-21 shows how to use a check box to edit items in a `ListView`. Figure 12-28 shows the resulting window after running the code. Select items using a mouse. Pressing the Print Selection button prints the selected items on the standard output.

### ***Listing 12-21.*** Using a Check Box to Edit ListView Items

```
// ListViewCheckBoxEditing.java
package com.jdojo.control;

import java.util.HashMap;
import java.util.Map;
import javafx.application.Application;
```

```

import javafx.beans.property.SimpleBooleanProperty;
import javafx.beans.value.ObservableValue;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.ListView;
import javafx.scene.control.SelectionMode;
import javafx.scene.control.cell.CheckBoxListCell;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import javafx.util.Callback;

public class ListViewCheckBoxEditing extends Application {
    Map<String, ObservableValue<Boolean>> map = new
HashMap<>();

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

    @Override
    public void start(Stage stage) {
        // Populate the map with ListView items as its keys
and
        // their selected state as the value
        map.put("Apple", new SimpleBooleanProperty(false));
        map.put("Banana", new SimpleBooleanProperty(false));
        map.put("Donut", new SimpleBooleanProperty(false));
        map.put("Hash Brown", new
SimpleBooleanProperty(false));

        ListView<String> breakfasts = new ListView<>();
        breakfasts.setPrefSize(200, 120);
        breakfasts.setEditable(true);
        breakfasts.getSelectionModel().setSelectionMode(Sele
ctionMode.MULTIPLE);

        // Add all keys from the map as items to the
ListView
        breakfasts.getItems().addAll(map.keySet());

        // Create a Callback object
        Callback<String, ObservableValue<Boolean>>
itemToBoolean =
            (String item) -> map.get(item);

        // Set the cell factory
        breakfasts.setCellFactory(CheckBoxListCell.forListVi
ew(itemToBoolean));

        Button printBtn = new Button("Print Selection");
        printBtn.setOnAction(e -> printSelection());

        VBox root = new VBox(new Label("Breakfasts:"),
breakfasts, printBtn);
    }
}

```

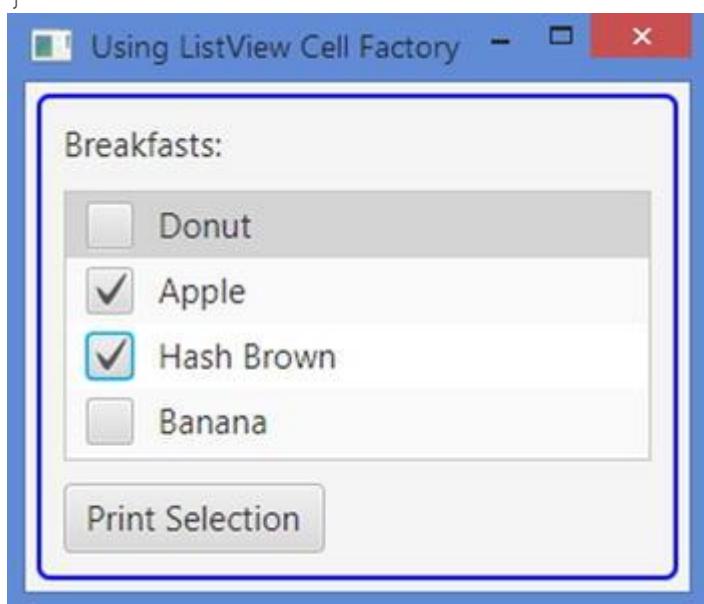
```

        root.setSpacing(10);
        root.setStyle("-fx-padding: 10;" +
                      "-fx-border-style: solid inside;" +
                      "-fx-border-width: 2;" +
                      "-fx-border-insets: 5;" +
                      "-fx-border-radius: 5;" +
                      "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using ListView Cell Factory");
        stage.show();
    }

    public void printSelection() {
        System.out.println("Selected items: ");
        for(String key: map.keySet()) {
            ObservableValue<Boolean> value = map.get(key);
            if (value.getValue()) {
                System.out.println(key);
            }
        }
        System.out.println();
    }
}

```



**Figure 12-28.** A ListView with a check box for editing its items

### Handling Events While Editing a ListView

An editable ListView fires three kinds of events:

- An `editStart` event when the editing starts
- An `editCommit` event when the edited value is committed

- An `editcancel` event when the editing is cancelled

The `ListView` class defines a `ListView.EditEvent<T>` static inner class to represent edit-related event objects. Its `getIndex()` method returns the index of the item that is edited.

The `getNewValue()` method returns the new input value.

The `getSource()` method returns the reference of the `ListView` firing the event. The `ListView` class provides `onEditStart`, `onEditCommit`, and `onEditCancel` properties to set the event handlers for these methods.

The following snippet of code adds an `editStart` event handler to a `ListView`. The handler prints the index that is being edited and the new item value:

```
ListView<String> breakfasts = new ListView<>();
...
breakfasts.setEditable(true);
breakfasts.setCellFactory(TextFieldListCell.forListView());

// Add an editStart event handler to the ListView
breakfasts.setOnEditStart(e ->
    System.out.println("Edit Start: Index="
+ e.getIndex() +
", item = " + e.getNewValue()));
```

**Listing 12-22** contains a complete program to show how to handle edit-related events in a `ListView`. Run the program and double-click an item to start editing. After changing the value, press Enter to commit editing or Esc to cancel editing. Edit-related event handlers print messages on the standard output.

### ***Listing 12-22.*** Handling Edit-Related Events in a `ListView`

```
// ListViewEditEvents.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.ListView;
import javafx.scene.control.cell.TextFieldListCell;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class ListViewEditEvents extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }
}
```

```

@Override
public void start(Stage stage) {
    ListView<String> breakfasts = new ListView<>();
    breakfasts.setPrefSize(200, 120);
    breakfasts.getItems().addAll("Apple", "Banana",
"Donut", "Hash Brown");
    breakfasts.setEditable(true);
    breakfasts.setCellFactory(TextFieldListCell.forListView());
}

// Add Edit-related event handlers
breakfasts.setOnEditStart(this::editStart);
breakfasts.setOnEditCommit(this::editCommit);
breakfasts.setOnEditCancel(this::editCancel);

HBox root = new HBox(new Label("Breakfast:"), 
breakfasts);
root.setSpacing(20);
root.setStyle("-fx-padding: 10;" +
"-fx-border-style: solid inside;" +
"-fx-border-width: 2;" +
"-fx-border-insets: 5;" +
"-fx-border-radius: 5;" +
"-fx-border-color: blue;");

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("Using ListView Edit Events");
stage.show();
}

public void editStart(ListView.EditEvent<String> e) {
    System.out.println("Edit Start: Index="
+ e.getIndex() +
", Item=" + e.getNewValue());
}

public void editCommit(ListView.EditEvent<String> e) {
    System.out.println("Edit Commit: Index="
+ e.getIndex() +
", Item=" + e.getNewValue());
}

public void editCancel(ListView.EditEvent<String> e) {
    System.out.println("Edit Cancel: Index="
+ e.getIndex() +
", Item=" + e.getNewValue());
}
}

```

## Styling *ListView* with CSS

The default CSS style-class name for a `ListView` is `list-view` and for `ListCell` it is `list-cell`. The `ListView` class has two CSS

pseudo-classes: horizontal and vertical. The `-fx-orientation` CSS property controls the orientation of the `ListView`, which can be set to *horizontal* or *vertical*.

You can style a `ListView` as you style any other controls. Each item is displayed in an instance of `ListCell`. `ListCell` provides several CSS pseudo-classes:

- `empty`
- `filled`
- `selected`
- `odd`
- `even`

The `empty` pseudo-class applies when the cell is empty.

The `filled` pseudo-class applies when the cell is not empty.

The `selected` pseudo-class applies when the cell is selected.

The `odd` and `even` pseudo-classes apply to cells with an odd and even index, respectively. The cell representing the first item is index 0 and it is considered an even cell.

The following CSS styles will highlight even cells with tan and odd cells with light gray:

```
.list-view .list-cell:even {  
    -fx-background-color: tan;  
}  
  
.list-view .list-cell:odd {  
    -fx-background-color: lightgray;  
}
```

Developers often ask how to remove the default alternate cell highlighting in a `ListView`. In the `modena.css` file, the default background color for all list cells is set to `-fx-control-inner-background`, which is a CSS-derived color. For all odd list cells, the default color is set to derive (`-fx-control-inner-background`, `-5%`). To keep the background color the same for all cells, you need to override the background color of odd list cells as follows:

```
.list-view .list-cell:odd {  
    -fx-background-color: -fx-control-inner-background;  
}
```

This only solves half of the problem; it only takes care of the background colors of the list cells in a normal state inside a `ListView`. A list cell can be in several states, for example, `focused`, `selected`, `empty`, or `filled`. To completely address this, you will need to set the appropriate background colors for list cells for all states. Please refer to

the `modena.css` file for a complete list of states that you will need to modify the background colors for list cells.

The `ListCell` class supports an `-fx-cell-size` CSS property that is the height of the cells in a vertical `ListView` and the width of cells in a horizontal `ListView`.

The list cell could be of the

type `ListCell`, `TextFieldListCell`, `ChoiceBoxListCell`, `ComboBoxListCell`, or `CheckBoxListCell`. The default CSS style-class names for subclasses of `ListCell` are `text-field-list-cell`, `choice-box-list-cell`, `combo-box-list-cell`, and `check-box-list-cell`. You can use these style class names to customize their appearance. The following CSS style will show the `TextField` in an editable `ListView` in yellow background:

```
.list-view .text-field-list-cell .text-field {
    -fx-background-color: yellow;
}
```

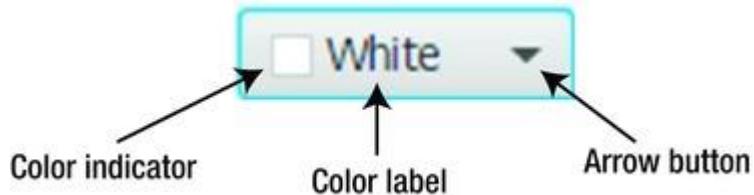
## Understanding the `ColorPicker` Control

`ColorPicker` is a combo box-style control that is especially designed for users to select a color from a standard color palette or create a color using a built-in color dialog. The `ColorPicker` class inherits from the `ComboBoxBase<Color>` class. Therefore, all properties declared in the `ComboBoxBase` class apply to the `ColorPicker` control as well. I have discussed several of these properties earlier in the “Understanding the `ComboBox` Control” section. If you want to know more about those properties, please refer to that section. For example, the `editable`, `onAction`, `showing`, and `value` properties work the same way in a `ColorPicker` as they do in a combo box.

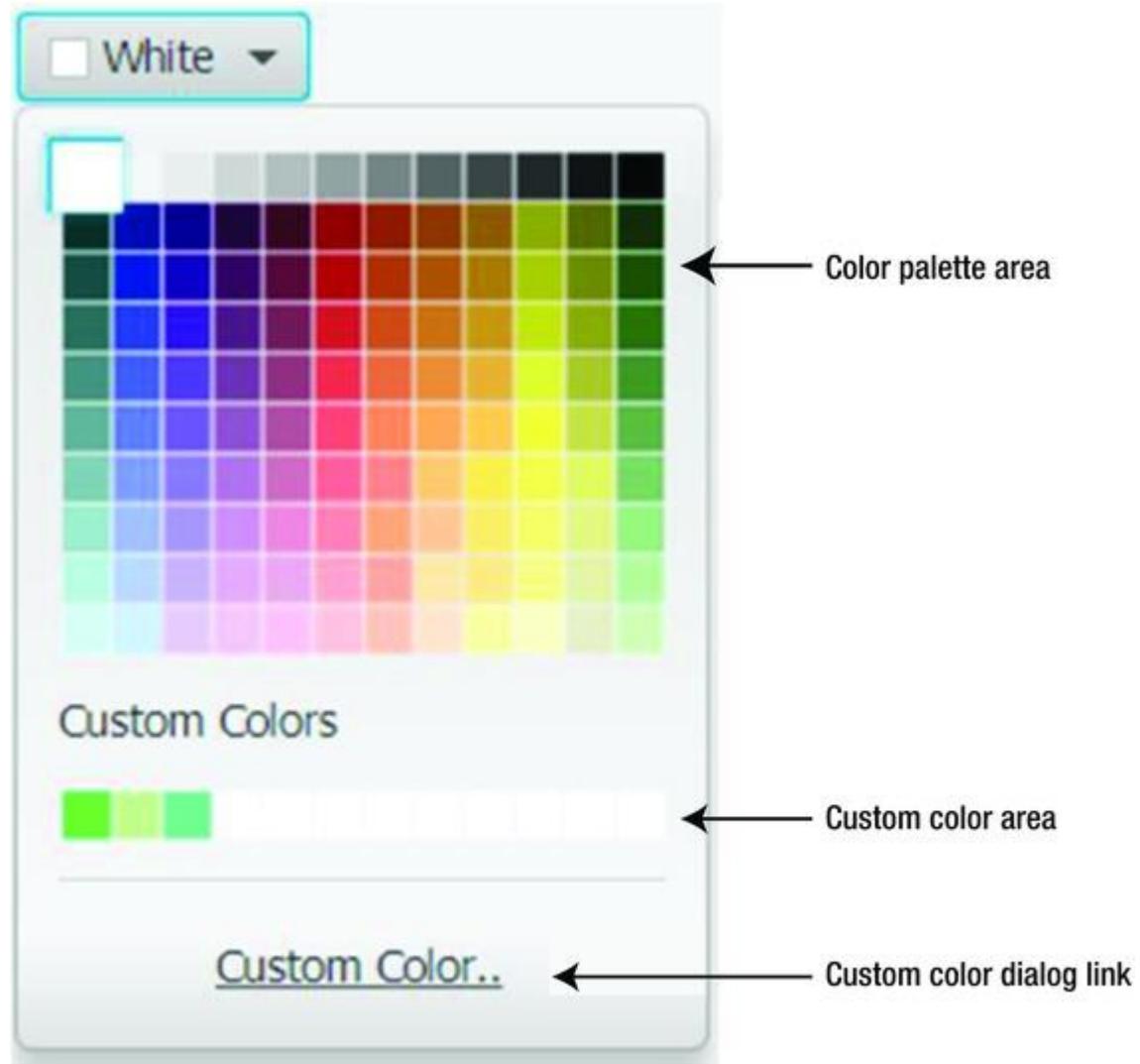
A `ColorPicker` has three parts:

- `ColorPicker` control
- Color palette
- Custom color dialog

A `ColorPicker` control consists of several components, as shown in Figure 12-29. You can customize their looks. The color indicator is a rectangle displaying the current color selection. The color label displays the color in text format. If the current selection is one of the standard colors, the label displays the color name. Otherwise, it displays the color value in hex format. Figure 12-30 shows a `ColorPicker` control and its color palette.



**Figure 12-29.** Components of a *ColorPicker* control



**Figure 12-30.** *ColorPicker* control and its color palette dialog box

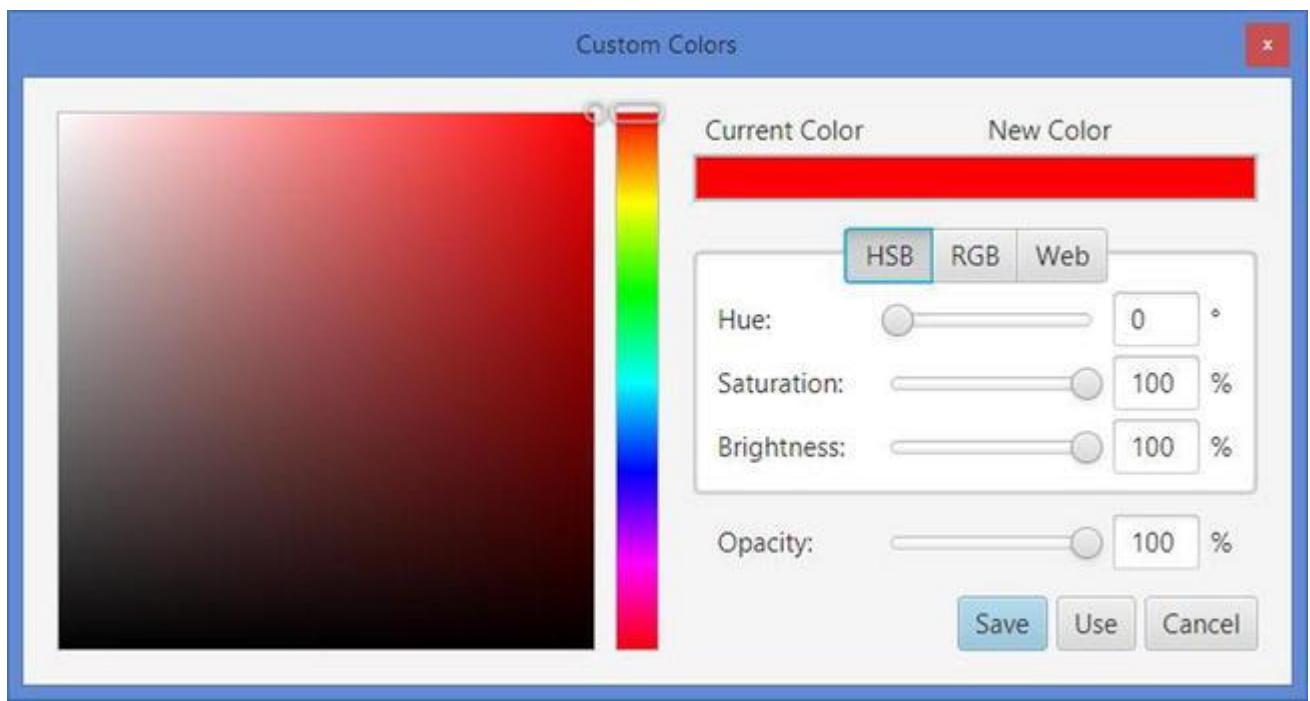
The color palette is shown as a pop-up when you click the arrow button in the control. The color palette consists of three areas:

- A color palette area to show a set of standard colors
- A custom colors area showing the list of custom colors
- A hyperlink to open the custom color dialog box

The color palette area shows a set of predefined standard colors. If you click one of the colors, it closes the pop-up and sets the selected color as the value for the `ColorPicker` control.

The custom color area shows a set of custom colors. When you open this pop-up for the first time, this area is absent. There are two ways to get colors in this area. You can load a set of custom colors or you can build and save custom colors using the custom color dialog box.

When you click the Custom Color... hyperlink, a custom color dialog box, as shown in Figure 12-31, is displayed. You can use HSB, RGB, or Web tab to build a custom color using one of these formats. You can also define a new color by selecting a color from the color area or the color vertical bar, which are on the left side of the dialog box. When you click the color area and the color bar, they show a small circle and rectangle to denote the new color. Clicking the Save button selects the custom color in the control and saves it to display later in the custom color area when you open the pop-up again. Clicking the Use button selects the custom color for the control.



**Figure 12-31.** Custom color dialog box of `ColorPicker`

### Using the `ColorPicker` Control

The `ColorPicker` class has two constructors. One of them is the default constructor and the other takes the initial color as an argument. The default constructor uses white as the initial color, as in the following code:

```
// Create a ColorPicker control with an initial color of white
ColorPicker bgColor1 = new ColorPicker();
```

```
// Create a ColorPicker control with an initial color of red
ColorPicker bgColor2 = new ColorPicker(Color.RED);
```

The value property of the control stores the currently selected color.

Typically, the value property is set when you select a color using the control. However, you can also set it directly in your code, as follows:

```
ColorPicker bgColor = new ColorPicker();
...
// Get the selected color
Color selectedColor = bgColor.getValue();

// Set the ColorPicker color to yellow
bgColor.setValue(Color.YELLOW);
```

The getCustomColors() method of the ColorPicker class returns a list of custom colors that you save in the custom colors dialog box. Note that custom colors are saved only for the current session and the current ColorPicker control. If you need to, you can save custom colors in a file or database and load them on startup. You will have to write some code to achieve this:

```
ColorPicker bgColor = new ColorPicker();
...
// Load two custom colors
bgColor.getCustomColors().addAll(Color.web("#07FF78"),
Color.web("#C2F3A7"));

...
// Get all custom colors
ObservableList<Color> customColors = bgColor.getCustomColors();
```

Typically, when a color is selected in a ColorPicker, you want to use the color for other controls. When a color is selected, the ColorPicker control generates an ActionEvent. The following snippet of code adds an ActionEvent handler to a ColorPicker. When a color is selected, the handler sets the new color as the fill color of a rectangle:

```
ColorPicker bgColor = new ColorPicker();
Rectangle rect = new Rectangle(0, 0, 100, 50);

// Set the selected color in the ColorPicker as the fill color of
// the Rectangle
bgColor.setOnAction(e -> rect.setFill(bgColor.getValue()));
```

The program in Listing 12-23 shows how to use ColorPicker controls. When you select a color using the ColorPicker, the fill color for the rectangle is updated.

### ***Listing 12-23.*** Using the ColorPicker Control

```
// ColorPickerTest.java
package com.jdojo.control;
```

```

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ColorPicker;
import javafx.scene.control.Label;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

public class ColorPickerTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        ColorPicker bgColor = new ColorPicker(Color.RED);

        // A Rectangle to show the selected color from the
        color picker
        Rectangle rect = new Rectangle(0, 0, 100, 50);
        rect.setFill(bgColor.getValue());
        rect.setStyle("-fx-stroke-width: 2; -fx-stroke:
black;");

        // Add an ActionEvent handler to the ColorPicker, so
you change
        // the fill color for the rectangle when you pick
a new color
        bgColor.setOnAction(e ->
rect.setFill(bgColor.getValue()));

        HBox root = new HBox(new Label("Color:"), bgColor,
rect);
        root.setSpacing(10);
        root.setStyle("-fx-padding: 10;" +
"-fx-border-style: solid inside;" +
"-fx-border-width: 2;" +
"-fx-border-insets: 5;" +
"-fx-border-radius: 5;" +
"-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using ColorPicker Controls");
        stage.show();
    }
}

```

The `ColorPicker` control supports three looks: combo-box look, button look, and split-button look. Combo-box look is the default look. Figure 12-32 shows a `ColorPicker` in these three looks, respectively.



**Figure 12-32.** Three looks of a `ColorPicker`

The `ColorPicker` class contains two string constants that are the CSS style-class name for the button and split-button looks. The constants are:

- `STYLE_CLASS_BUTTON`
- `STYLE_CLASS_SPLIT_BUTTON`

If you want to change the default look of a `ColorPicker`, add one of the above constants as its style class, as follows:

```
// Use default combo-box look
ColorPicker cp = new ColorPicker(Color.RED);

// Change the look to button
cp.getStyleClass().add(ColorPicker.STYLE_CLASS_BUTTON);

// Change the look to split-button
cp.getStyleClass().add(ColorPicker.STYLE_CLASS_SPLIT_BUTTON);
```

**Tip** It is possible to add both `STYLE_CLASS_BUTTON` and `STYLE_CLASS_SPLIT_BUTTON` as style classes for a `ColorPicker`. In such a case, the `STYLE_CLASS_BUTTON` is used.

### Styling `ColorPicker` with CSS

The default CSS style-class name for a `ColorPicker` is `color-picker`. You can style almost every part of a `ColorPicker`, for example, color indicator, color label, color palette dialog, and custom color dialog. Please refer to the `modena.css` file for complete reference.

The `-fx-color-label-visible` CSS property of the `ColorPicker` sets whether the color label is visible or not. Its default value is true. The following code makes the color label invisible:

```
.color-picker {
    -fx-color-label-visible: false;
}
```

The color indicator is a rectangle, which has a style class name of `picker-color-rect`. The color label is a `Label`, which has a style class name of `color-picker-label`. The following code shows the color label in blue and sets a 2px thick black stroke around the color indicator rectangle:

```
.color-picker .color-picker-label {
    -fx-text-fill: blue;
}

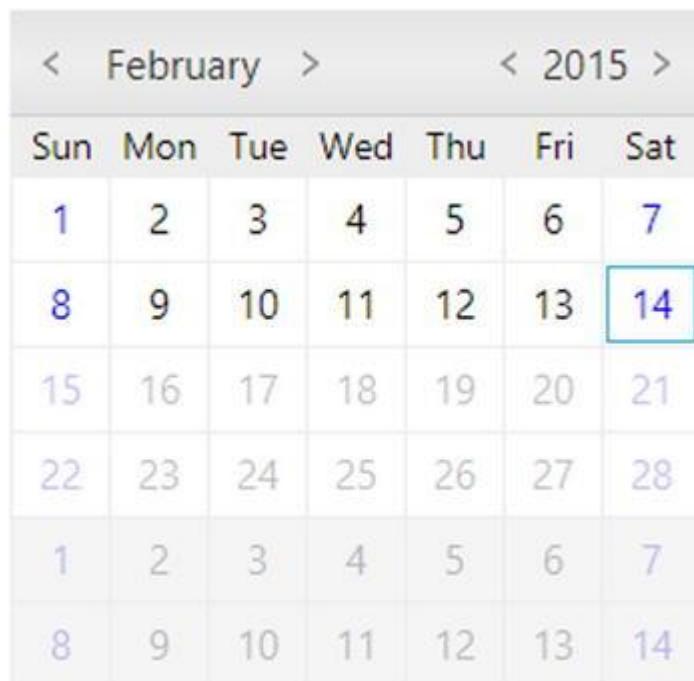
.color-picker .picker-color .picker-color-rect {
    -fx-stroke: black;
    -fx-stroke-width: 2;
}
```

The style class name for the color palette is `color-palette`. The following code hides the Custom Colors... hyperlink on the color palette:

```
.color-palette .hyperlink {
    visibility: hidden;
}
```

## Understanding the *DatePicker* Control

`DatePicker` is a combo-box style control. The user can enter a date as text or select a date from a calendar. The calendar is displayed as a pop-up for the control, as shown in Figure 12-33. The `DatePicker` class inherits from the `ComboBoxBase<LocalDate>` class. All properties declared in the `ComboBoxBase` class are also available to the `DatePicker` control.



**Figure 12-33.** Calendar pop-up for a `DatePicker` control

The first row of the pop-up displays the month and year. You can scroll through months and years using the arrows. The second row displays the short names of weeks. The first column displays the week number of the year. By default, the week numbers column is not displayed. You can use

the context menu on the pop-up to display it or you can set the `showWeekNumbers` property of the control to show it.

The calendar always displays dates for 42 days. Dates not applicable to the current month are disabled for selection. Each day cell is an instance of the `DateCell` class. You can provide a cell factory to use your custom cells. You will have an example of using a custom cell factory later.

Right-clicking the first row, week names, week number column, or disabled dates displays the context menu. The context menu also contains a Show Today menu item, which scrolls the calendar to the current date.

## Using the *DatePicker* Control

You can create a `DatePicker` using its default constructor; it uses `null` as the initial value. You can also pass a `LocalDate` to another constructor as the initial value, as in the following code:

```
// Create a DatePicker with null as its initial value
DatePicker birthDate1 = new DatePicker();

// Use September 19, 1969 as its initial value
DatePicker birthDate2 = new DatePicker(LocalDate.of(1969, 9,
19));
```

The `value` property of the control holds the current date in the control. You can use the property to set a date. When the control has a `null` value, the pop-up shows the dates for the current month. Otherwise, the pop-up shows the dates of the month of the current value, as with the following code:

```
// Get the current value
LocalDate dt = birthDate.getValue();

// Set the current value
birthDate.setValue(LocalDate.of(1969, 9, 19));
```

The `DatePicker` control provides a `TextField` to enter a date as text. Its `editor` property stores the reference of the `TextField`. The property is read-only. If you do not want users to enter a date, you can set the `editable` property of the `DatePicker` to false, as in the following code:

```
DatePicker birthDate = new DatePicker();

// Users cannot enter a date. They must select one from the
// popup.
birthDate.setEditable(false);
```

`DatePicker` has a `converter` property that uses a `StringConverter` to convert a `LocalDate` to a string and vice versa. Its `value` property stores the date as `LocalDate` and its `editor` displays it as a string, which is the formatted date. When you enter a date as text,

the converter converts it to a `LocalDate` and stores it in the `value` property. When you pick a date from the calendar pop-up, the converter creates a `LocalDate` to store in the `value` property and it converts it to a string to display in the editor. The default converter uses the default `Locale` and chronology to format the date. When you enter a date as text, the default converter expects the text in the default `Locale` and chronology format.

**Listing 12-24** contains the code for a `LocalDateStringConverter` class that is a `StringConverter` for `LocalDate`. By default, it formats dates in `MM/dd/yyyy` format. You can pass a different format in its constructor.

### ***Listing 12-24.*** A `StringConverter` to Convert a `LocalDate` to a String and Vice Versa

```
// LocalDateStringConverter.java
package com.jdojo.control;

import javafx.util.StringConverter;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;

public class LocalDateStringConverter extends
StringConverter<LocalDate> {
    private String pattern = "MM/dd/yyyy";
    private DateTimeFormatter dtFormatter;

    public LocalDateStringConverter() {
        dtFormatter = DateTimeFormatter.ofPattern(pattern);
    }

    public LocalDateStringConverter(String pattern) {
        this.pattern = pattern;
        dtFormatter = DateTimeFormatter.ofPattern(pattern);
    }

    @Override
    public LocalDate fromString(String text) {
        LocalDate date = null;
        if (text != null && !text.trim().isEmpty()) {
            date = LocalDate.parse(text, dtFormatter);
        }
        return date;
    }

    @Override
    public String toString(LocalDate date) {
        String text = null;
        if (date != null) {
```

```

        text = dtFormatter.format(date);
    }
    return text;
}
}

```

To format the date in "MMMM dd, yyyy" format, for example, May 29, 2013, you would create and set the converter as follows:

```

DatePicker birthDate = new DatePicker();
birthDate.setConverter(new LocalDateStringConverter("MMMM dd,
yyyy"));

```

You can configure the DatePicker control to work with a specific chronology instead of the default one. The following statement sets the chronology to Thai Buddhist chronology:

```
birthDate.setChronology(ThaiBuddhistChronology.INSTANCE);
```

You can change the default Locale for the current instance of the JVM and the DatePicker will use the date format and chronology for the default Locale:

```
// Change the default Locale to Canada
Locale.setDefault(Locale.CANADA);
```

Each day cell in the pop-up calendar is an instance of the DateCell class, which is inherited from

the Cell<LocalDate> class. The dayCellFactory property of the DatePicker class lets you provide a custom day cell factory. The concept is the same as discussed earlier for providing the cell factory for the ListView control. The following statement creates a day cell factory. It changes the text color of weekend cells to blue and disables all future day cells. If you set this day cell factory to a DatePicker, the pop-up calendar will not let users select a future date because you will have disabled all future day cells:

```

Callback<DatePicker, DateCell> dayCellFactory =
    new Callback<DatePicker, DateCell>() {
        public DateCell call(final DatePicker datePicker) {
            return new DateCell() {
                @Override
                public void updateItem(LocalDate item,
boolean empty) {
                    // Must call super
                    super.updateItem(item, empty);

                    // Disable all future date cells
                    if (item.isAfter(LocalDate.now()))
{
                        this.setDisable(true);
}

                    // Show Weekends in blue
                    DayOfWeek day
= DayOfWeek.from(item);

```

```
        if (day == DayOfWeek.SATURDAY ||  
            day == DayOfWeek.SUNDAY) {  
            his.setTextFill(Color.BLUE)  
;  
        }  
    }  
};  
}  
};  
};
```

The following snippet of code sets a custom day cell factory for a birth date `DatePicker` control. It also makes the control noneditable. The control will force the user to select a nonfuture date from the pop-up calendar:

```
DatePicker birthDate = new DatePicker();  
  
// Set a day cell factory to disable all future day cells  
// and show weekends in blue  
birthDate.setDayCellFactory(dayCellFactory);  
  
// Users must select a date from the popup calendar  
birthDate.setEditable(false);
```

The `DatePicker` control fires an `ActionEvent` when its `value` property changes. The `value` property may change when a user enters a date, selects a date from the pop-up, or a date is set programmatically, as provided in the following code:

```
// Add an ActionEvent handler  
birthDate.setOnAction(e -> System.out.println("Date changed to:  
+ birthDate.getValue()));
```

**Listing 12-25** has a complete program showing how to use a `DatePicker` control. It uses most of the features of the `DatePicker`. It displays a window as shown in Figure 12-34. The control is noneditable, forcing the user to select a nonfuture date from the pop-up.

**Listing 12-25.** Using the DatePicker Control

```
// DatePickerTest.java
package com.jdojo.control;

import java.time.DayOfWeek;
import java.time.LocalDate;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.DateCell;
import javafx.scene.control.DatePicker;
import javafx.scene.control.Label;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.util.Callback;
```

```

public class DatePickerTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    public void start(Stage stage) {
        DatePicker birthDate = new DatePicker();
        birthDate.setEditable(false);

        // Print the new date on standard output
        birthDate.setOnAction(e ->
            System.out.println("New Date:" +
+ birthDate.getValue()));

        String pattern = "MM/dd/yyyy";
        birthDate.setConverter(new
LocalDateStringConverter(pattern));
        birthDate.setPromptText(pattern.toLowerCase());

        // Create a day cell factory
        Callback<DatePicker, DateCell> dayCellFactory =
        new Callback<DatePicker, DateCell>() {
            public DateCell call(final DatePicker
datePicker) {
                return new DateCell() {
                    @Override
                    public void updateItem(LocalDate
item, boolean empty) {
                        // Must call super
                        super.updateItem(item,
empty);

                        // Disable all future date
                        cells
                        if
                            (item.isAfter(LocalDate.now())) {
                                this.setDisable(true)
                            }
                        }

                        // Show Weekends in blue
                        color
                        DayOfWeek day
                        = DayOfWeek.from(item);
                        if (day ==
DayOfWeek.SATURDAY ||
DayOfWeek.SUNDAY) {
                            day ==
                            this.setTextFill(Colo
r.BLUE);
                        }
                    }
                };
            }
        };
    }
}

```

```

        // Set the day cell factory
        birthDate.setDayCellFactory(dayCellFactory);

        HBox root = new HBox(new Label("Birth Date:"),
birthDate);
        root.setStyle("-fx-padding: 10;" +
                  "-fx-border-style: solid inside;" +
                  "-fx-border-width: 2;" +
                  "-fx-border-insets: 5;" +
                  "-fx-border-radius: 5;" +
                  "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using DatePicker Control");
        stage.show();
        stage.sizeToScene();
    }
}

```



**Figure 12-34.** A DatePicker control to select a nonfuture date

### Styling DatePicker with CSS

The default CSS style-class name for a DatePicker is `date-picker`, and for its pop-up, the class name is `date-picker-popup`. You can style almost every part of a DatePicker, for example, the month-year pane in the top area of the pop-up, day cells, week number cells, and current day cell. Please refer to the `modena.css` file for complete reference.

The CSS style-class name for day cell is `day-cell`. The day cell for the current date has the style-class name as `today`. The following styles display the current day number in bold and all day numbers in blue:

```

/* Display current day numbers in bolder font */
.date-picker-popup > * > .today {
    -fx-font-weight: bolder;
}

/* Display all day numbers in blue */
.date-picker-popup > * > .day-cell {
    -fx-text-fill: blue;
}

```

## Understanding Text Input Controls

JavaFX supports text input controls that let users work with single line or multiple lines of plain text. I will discuss `TextField`, `PasswordField`, and `TextArea` text input controls in this section. All text input controls are inherited from the `TextInputControl` class. Please refer to Figure 12-1 for a class diagram for the text input controls.

**Tip** JavaFX provides a rich text edit control named `HTMLEditor`. I will discuss `HTMLEditor` later in this chapter.

The `TextInputControl` class contains the properties and methods that apply to all types of text input controls. Properties and methods related to the current caret position and movement and text selection are in this class. Subclasses add properties and methods applicable to them. Table 12-5 lists the properties declared in the `TextInputControl` class.

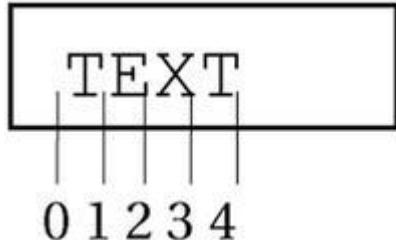
**Table 12-5.** Properties Declared in the `TextInputControl` Class

Property	Type	Description
anchor	<code>ReadOnlyIntegerProperty</code>	It is the anchor of the text selected at the opposite end of the caret in the selection.
caretPosition	<code>ReadOnlyIntegerProperty</code>	It is the current position of the caret within the text.
editable	<code>BooleanProperty</code>	It is true if the control is editable. Otherwise, it is false.
font	<code>ObjectProperty&lt;Font&gt;</code>	It is the default font for the control.
length	<code>ReadOnlyIntegerProperty</code>	It is the number of characters in the control.
promptText	<code>StringProperty</code>	It is the prompt text. It is displayed in the control when control has no text.

Property	Type	Description
selectedText	ReadOnlyStringProperty	It is the selected text in the control.
selection	ReadOnlyObjectProperty<IndexRange>	It is the selected text index range.
text	StringProperty	It is the text in the control.

### Positioning and Moving Caret

All text input controls provide a caret. By default, a caret is a blinking vertical line when the control has focus. The current caret position is the target for the next input character from the keyboard. The caret position starts at zero, which is before the first character. Position 1 is after the first character and before the second character and so on. Figure 12-35 shows the caret positions in a text input control that has four characters. The number of characters in the text determines the valid range for the caret position, which is zero to the length of the text. Zero is the only valid caret position if the control does not contain text.



**Figure 12-35.** Caret positions in a text input control having four characters

Several methods take a caret position as an argument. Those methods clamp the argument value to the valid caret position range. Passing a caret position outside the valid range will not throw an exception. For example, if the control has four characters and you want to move the caret to position 10, the caret will be positioned at position 4.

The read-only `caretPosition` property contains the current caret position. Use the `positionCaret(int pos)` method to position the caret at the specified `pos`. The `backward()` and `forward()` methods move the caret one character backward and forward, respectively, if there is no selection. If there is a selection, they move the caret position to the beginning and end and clear the selection.

The `home()` and `end()` methods move the caret before the first character and after the last character, respectively, and clear the

selection. The `nextWord()` method moves the caret to the beginning of the next word and clears the selection. The `endOfNextWord()` method moves the caret to the end of the next word and clears the selection. The `previousWord()` method moves the caret to the beginning of the previous word and clears the selection.

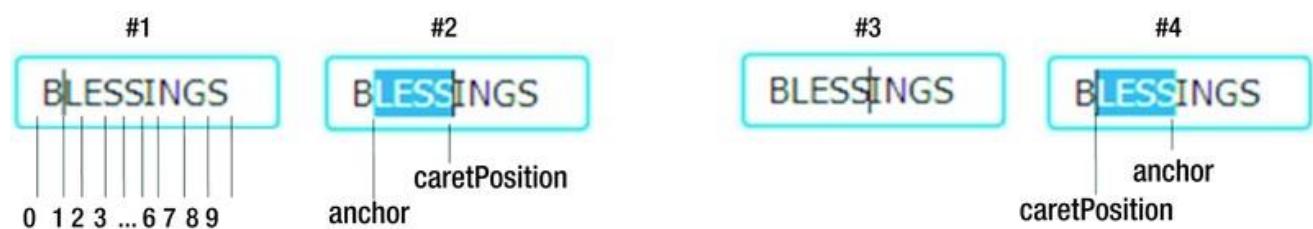
## Making Text Selection

The `TextInputControl` class provides a rich API through its properties and methods to deal with text selection. Using the selection API, you can select the entire or partial text and get the selection information.

The `selectedText` property contains the value of the selected text. Its value is an empty string if there is no selection.

The `selection` property contains an `IndexRange` that holds the index range of the selection. The `getStart()` and `getEnd()` methods of the `IndexRange` class return the start index and end index of the selection, respectively, and its `getLength()` method returns the length of the selection. If there is no selection, the lower and upper limits of the range are the same and they are equal to the `caretPosition` value.

The `anchor` and `caretPosition` properties play a vital role in text selection. The value of these properties defines the selection range. The same value for both properties indicates no selection. Either property may indicate the start or end of the selection range. The `anchor` value is the caret position when the selection started. You can select characters by moving the caret backward or forward. For example, you can use the left or right arrow key with the Shift key pressed to select a range of characters. If you move the caret forward during the selection process, the `anchor` value will be less than the `caretPosition` value. If you move the caret backward during the selection process, the `anchor` value will be greater than the `caretPosition` value. Figure 12-36 shows the relation between the `anchor` and `caretPosition` values.



**Figure 12-36.** Relation between the `anchor` and `caretPosition` properties of a text input control

In Figure 12-36, the part labeled #1 shows a text input control with the text BLESSINGS. The `caretPosition` value is 1. The user selects four

characters by moving the caret four positions forward, for example, by pressing Shift key and right arrow key or by dragging the mouse. The selectedText property, as shown in the part labeled #2, is LESS. The anchor value is 1 and the caretPosition value is 5. The selection property has an IndexRange of 1 to 5.

In the part labeled #3, the caretPosition value is 5. The user selects four characters by moving the caret backward as shown in the part labeled #4. The selectedText property, as shown in part labeled #4, is LESS. The anchor value is 5 and the caretPosition value is 1. The selection property has an IndexRange of 1 to 5. Notice that in the parts labeled #2 and #4, the anchor and caretPosition values are different and the selectedText and selection properties are the same. Apart from the selection properties, the TextInputControl contains several useful selection-related methods:

- selectAll()
- deselect()
- selectRange(int anchor, int caretPosition)
- selectHome()
- selectEnd()
- extendSelection(int pos)
- selectBackward()
- selectForward()
- selectPreviousWord()
- selectEndOfNextWord()
- selectNextWord()
- selectPositionCaret(int pos)
- replaceSelection(String replacement)

Notice that you have a positionCaret(int pos) method and a selectPositionCaret(int pos) method. The former positions the caret at the specified position and clears the selection. The latter moves the caret to the specified pos and extends the selection if one exists. If no selection exists, it forms a selection by the current caret position as the anchor and moving the caret to the specified pos.

The replaceSelection(String replacement) method replaces the selected text by the specified replacement. If there is no selection, it clears the selection and inserts the specified replacement at the current caret position.

## Modifying the Content

The `text` property of the `TextInputControl` class represents the textual content of text input controls. You can change the content using the `setText(String text)` method and get it using the `getText()` method. The `clear()` method sets the content to an empty string.

The `insertText(int index, String text)` method inserts the specified text at the specified index. It throws an `IndexOutOfBoundsException` if the specified index is outside the valid range (zero to the length of the content).

The `appendText(String text)` method appends the specified text to the content. The `deleteText()` method lets you delete a range of characters from the content. You can specify the range as an `IndexRange` object or start and end index.

The `deleteNextChar()` and `deletePreviousChar()` methods delete the next and previous character, respectively, from the current caret position if there is no selection. If there is a selection, they delete the selection. They return `true` if the deletion was successful. Otherwise, they return `false`.

The read-only `length` property represents the length of the content. It changes as you modify the content. Practically, the `length` value can be very big. There is no direct way to restrict the number of characters in a text input control. I will cover an example of restricting the length of text shortly.

## Cutting, Copying, and Pasting Text

The text input controls supports cut, copy, and paste features programmatically, using the mouse and keyboard. To use these features using the mouse and keyboard, use the standard steps supported on your platform. Use the `cut()`, `copy()`, and `paste()` methods to use these features programmatically. The `cut()` method transfers the currently selected text to the clipboard and removes the current selection.

The `copy()` method transfers the currently selected text to the clipboard without removing the current selection. The `paste()` method replaces the current selection with the content in the clipboard. If there is no selection, it inserts the clipboard content at the current caret position.

## An Example

The program in Listing 12-26 demonstrates how the different properties of text input control change. It displays a window as shown in Figure 12-37. The program uses a `TextField`, which is a text input control, to display one line of text. Each property is displayed in a `Label` by binding

the `textProperties` to the properties of the `TextField`. After running the program, change the text in the name field, move the caret, and change the selection to see how the properties of the `TextField` change.

### ***Listing 12-26.*** Using the Properties of Text Input Controls

```
// TextControlProperties.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class TextControlProperties extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        TextField nameFld = new TextField();
        Label anchorLbl = new Label("");
        Label caretLbl = new Label("");
        Label lengthLbl = new Label("");
        Label selectedTextLbl = new Label("");
        Label selectionLbl = new Label("");
        Label textLbl = new Label("");

        // Bind text property of the Labels to the
        // properties of the TextField
        anchorLbl.textProperty().bind(nameFld.anchorProperty()
            .asString());
        caretLbl.textProperty().bind(nameFld.caretPositionPr
        operty().asString());
        lengthLbl.textProperty().bind(nameFld.lengthProperty
            .asString());
        selectedTextLbl.textProperty().bind(nameFld.selected
        TextProperty());
        selectionLbl.textProperty().bind(nameFld.selectionPr
        operty().asString());
        textLbl.textProperty().bind(nameFld.textProperty());

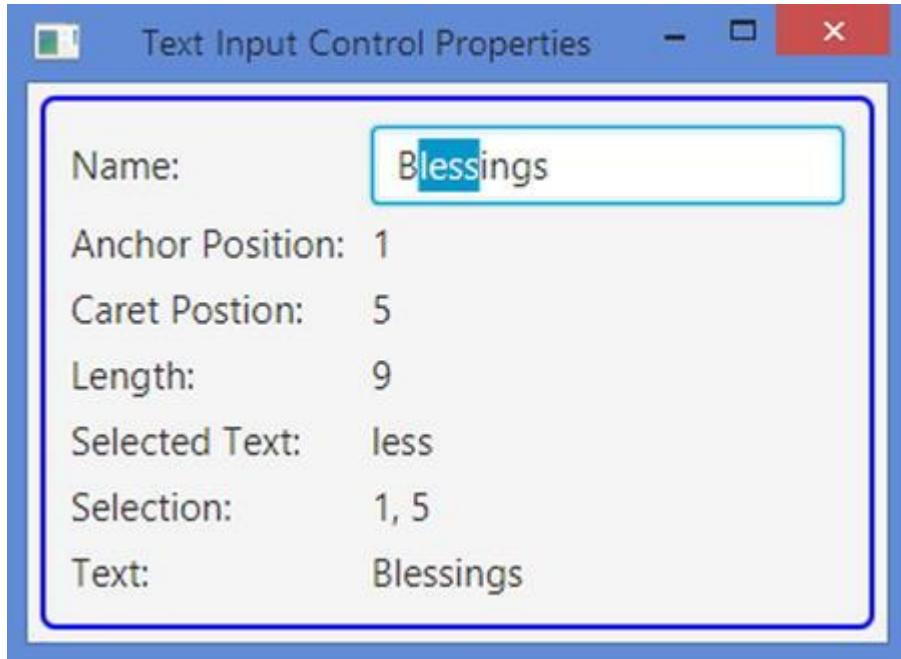
        GridPane root = new GridPane();
        root.setHgap(10);
        root.setVgap(5);
        root.addRow(0, new Label("Name:"), nameFld);
        root.addRow(1, new Label("Anchor Position:"),
        anchorLbl);
        root.addRow(2, new Label("Caret Postion:"),
```

```

caretLbl);
        root.addRow(3, new Label("Length:"), lengthLbl);
        root.addRow(4, new Label("Selected Text:"),
selectedTextLbl);
        root.addRow(5, new Label("Selection:"), selectionLbl);
        root.addRow(6, new Label("Text:"), textLbl);
        root.setStyle("-fx-padding: 10;" +
                      "-fx-border-style: solid inside;" +
                      "-fx-border-width: 2;" +
                      "-fx-border-insets: 5;" +
                      "-fx-border-radius: 5;" +
                      "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Text Input Control Properties");
        stage.show();
    }
}

```



**Figure 12-37.** Using properties of text input controls

### Styling *TextInputControl* with CSS

The `TextInputControl` class introduces a CSS pseudo-class named `readonly`, which applies when the control is not editable. It adds the following style properties:

- `-fx-font`
- `-fx-text-fill`
- `-fx-prompt-text-fill`

- `-fx-highlight-fill`
- `-fx-highlight-text-fill`
- `-fx-display-caret`

The `-fx-font` property is inherited from the parent by default. The value for the `-fx-display-caret` property could be true or false. When it is true, the caret is displayed when the control has focus. Otherwise, the caret is not displayed. Its default value is true. Most of the other properties affect background and text colors.

## Understanding the *TextField* Control

*TextField* is a text input control. It inherits from the `TextInputControl` class. It lets the user enter a single line of plain text. If you need a control to enter multiline text, use `TextArea` instead. Newline and tab characters in the text are removed. Figure 12-38 shows a window with two `TextFields` having the text Layne and Estes.



**Figure 12-38.** A window with two *TextField* controls

You can create a `TextField` with an empty initial text or with a specified initial text, as shown in the following code:

```
// Create a TextField with an empty string as initial text
TextField nameFld1 = new TextField();

// Create a TextField with "Layne Estes" as an initial text
TextField nameFld2 = new TextField("Layne Estes");
```

As I have already mentioned, the `text` property of the `TextField` stores the textual content. If you are interested in handling the changes in a `TextField`, you need to add a `ChangeListener` to its `text` property. Most of the time you will be using its `setText(String newText)` method to set new text and the `getText()` method to get the text from it. `TextField` adds the following properties:

- `alignment`

- `onAction`
- `prefColumnCount`

The `alignment` property determines the alignment of the text within the `TextField` area when there is empty space. Its default value is `CENTER_LEFT` if the node orientation is `LEFT_TO_RIGHT` and `CENTER_RIGHT` if the node orientation is `RIGHT_TO_LEFT`. The `onAction` property is an `ActionEvent` handler, which is called when the Enter key is pressed in the `TextField`, as shown in the following code:

```
TextField nameFld = new TextField();  
nameFld.setOnAction(e -> /* Your ActionEvent handler code... */ );
```

The `prefColumnCount` property determines the width of the control. By default, its value is 12. A column is wide enough to display an uppercase letter W. If you set its value to 10, the `TextField` will be wide enough to display ten letter Ws, as shown in the following code:

```
// Set the preferred column count to 10  
nameFld.setPrefColumnCount(10);
```

`TextField` provides a default context menu, as shown in Figure 12-39, that can be displayed by clicking the right mouse button. Menu items are enabled or disabled based on the context. You can replace the default context menu with a custom context menu. Currently, there is no way to customize the default context menu.



**Figure 12-39.** The default context menu for *TextField*

The following snippet of code sets a custom context menu for a *TextField*. It displays a menu item stating that the context menu is disabled. Selecting the menu item does nothing. You will need to add an *ActionEvent* handler to the menu items in context menu to perform some action.

```
ContextMenu cm = new ContextMenu();
MenuItem dummyItem = new MenuItem("Context menu is disabled");
cm.getItems().add(dummyItem);

TextField nameFld = new TextField();
nameFld.setContextMenu(cm);
```

The program in **Listing 12-27** shows how to use *TextField* controls. It displays two *TextFields*. It shows adding *ActionEvent* handlers, a custom context menu, and *ChangeListeners* added to *TextFields*.

### **Listing 12-27.** Using the *TextField* Control

```
// TextFieldTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.beans.value.ObservableValue;
import javafx.scene.Scene;
import javafx.scene.control.ContextMenu;
import javafx.scene.control.Label;
import javafx.scene.control.MenuItem;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class TextFieldTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    public void start(Stage stage) {
        // Create a TextFiled with an empty string as its
        // initial text
        TextField firstNameFld = new TextField();
        TextField lastNameFld = new TextField();

        // Both fields should be wide enough to display 15
        // chars
        firstNameFld.setPrefColumnCount(15);
        lastNameFld.setPrefColumnCount(15);

        // Add a ChangeListener to the text property
        firstNameFld.textProperty().addListener(this::change
d);
```

```

lastNameFld.textProperty().addListener(this::changed)
);

// Add a dummy custom context menu for the firstname
field
ContextMenu cm = new ContextMenu();
MenuItem dummyItem = new MenuItem("Context menu is
disabled");
cm.getItems().add(dummyItem);
firstNameFld.setContextMenu(cm);

// Set ActionEvent handlers for both fields
firstNameFld.setOnAction(e -> nameChanged("First
Name"));
lastNameFld.setOnAction(e -> nameChanged("Last
Name"));

GridPane root = new GridPane();
root.setHgap(10);
root.setVgap(5);
root.addRow(0, new Label("First Name:"), 
firstNameFld);
root.addRow(1, new Label("Last Name:"), 
lastNameFld);
root.setStyle("-fx-padding: 10;" +
"-fx-border-style: solid inside;" +
"-fx-border-width: 2;" +
"-fx-border-insets: 5;" +
"-fx-border-radius: 5;" +
"-fx-border-color: blue;");

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("Using TextField Controls");
stage.show();
}

public void nameChanged(String fieldName) {
    System.out.println("Action event fired on "
+ fieldName);
}

public void changed(ObservableValue<? extends String> prop,
                    String oldValue,
                    String newValue) {
    System.out.println("Old = " + oldValue + ", new = "
+ newValue);
}
}

```

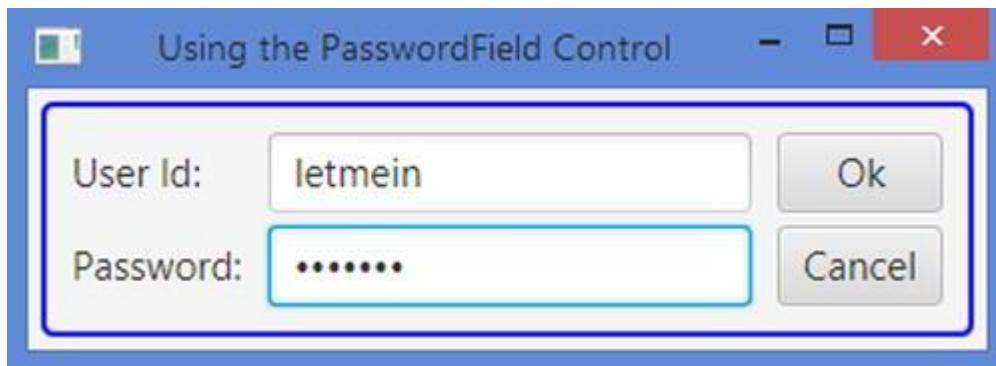
## Styling *TextField* with CSS

The default CSS style-class name for a *TextField* is `text-field`. It adds an `-fx-alignment` property that is the alignment of its text within

its content area. There is nothing special that needs to be said about styling `TextField`.

## Understanding the `PasswordField` Control

`PasswordField` is a text input control. It inherits from `TextField` and it works much the same as `TextField` except it masks its text, that is, it does not display the actual characters entered. Rather, it displays an echo character for each character entered. The default echo character is a bullet. Figure 12-40 shows a window with a `PasswordField`.



**Figure 12-40.** A window using a `PasswordField` control

The `PasswordField` class provides only one constructor, which is a no-args constructor. You can use the `setText()` and `getText()` methods to set and get, respectively, the actual text in a `PasswordField`, as in the following code. Typically, you do not set the password text. The user enters it.

```
// Create a PasswordField
PasswordField passwordFld = new PasswordField();
...
// Get the password text
String passStr = passwordFld.getText();
```

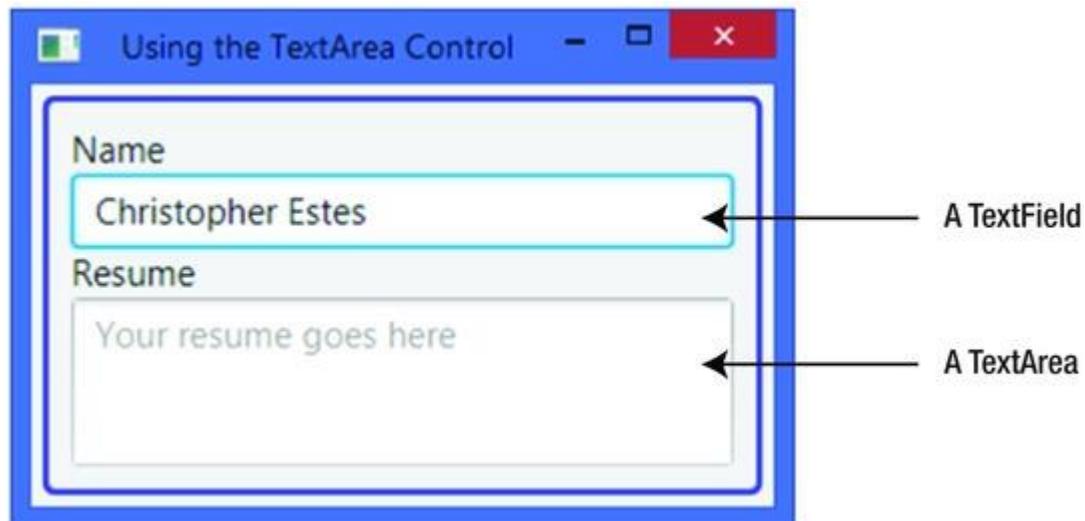
The `PasswordField` overrides the `cut()` and `copy()` methods of the `TextInputControl` class to make them no-op methods. That is, you cannot transfer the text in a `PasswordField` to the clipboard using the keyboard shortcuts or the context menu.

The default CSS style-class name for a `PasswordField` is `password-field`. It has all of the style properties of `TextField`. It does not add any style properties.

## Understanding the `TextArea` Control

`TextArea` is a text input control. It inherits from the `TextInputControl` class. It lets the user enter multiline plain text.

If you need a control to enter a single line of plain text, use `TextField` instead. If you want to use rich text, use the `HTMLEditor` control. Unlike the `TextField`, newline and tab characters in the text are preserved. A newline character starts a new paragraph in a `TextArea`. Figure 12-41 shows a window with a `TextField` and a `TextArea`. The user can enter a multiline résumé in the `TextArea`.



**Figure 12-41.** A window with a `TextArea` control

You can create a `TextArea` with an empty initial text or with a specified initial text using the following code:

```
// Create a TextArea with an empty string as its initial text
TextArea resume1 = new TextArea();

// Create a TextArea an initial text
TextArea resume2 = new TextArea("Years of Experience: 19");
```

As already discussed in the previous section, the `text` property of the `TextArea` stores the textual content. If you are interested in handling the changes in a `TextArea`, you need to add a `ChangeListener` to its `text` property. Most of the time, you will be using its `setText(String newText)` method to set new text and its `getText()` method to get the text from it.

`TextArea` adds the following properties:

- `prefColumnCount`
- `prefRowCount`
- `scrollLeft`
- `scrollTop`
- `wrapText`

The `prefColumnCount` property determines the width of the control. By default, its value is 32. A column is wide enough to display an uppercase letter W. If you set its value to 80, the `TextArea` will be wide enough to display 80 letter Ws. The following code accomplishes this:

```
// Set the preferred column count to 80
resume1.setPrefColumnCount(80);
```

The `prefRowCount` property determines the height of the control. By default, it is 10. The following code sets the row count to 20:

```
// Set the preferred row count to 20
resume.setPrefColumnCount(20);
```

If the text exceeds the number of columns and rows, the horizontal and vertical scroll panes are automatically displayed.

Like `TextField`, `TextArea` provides a default context menu. Please refer the “Understanding Text Input Controls” section for more detail on how to customize the default context menu.

The `scrollLeft` and `scrollTop` properties are the number of pixels that the text is scrolled to at the top and left. The following code sets it to 30px:

```
// Scroll the resume text by 30px to the top and 30 px to the
left
resume.scrollTop(30);
resume.scrollLeft(30);
```

By default, `TextArea` starts a new line when it encounters a newline character in its text. A newline character also creates a new paragraph except for the first paragraph. By default, the text is not wrapped to the next line if it exceeds the width of the control.

The `wrapText` property determines whether the text is wrapped to another line when its run exceeds the width of the control. By default, its value is false. The following code would set the default to true:

```
// Wrap the text if needed
resume.setWrapText(true);
```

The `getParagraphs()` method of the `TextArea` class returns an unmodifiable list of all paragraphs in its text. Each element in the list is a paragraph, which is an instance of `CharSequence`. The returned paragraph does not contain the newline characters. The following snippet of code prints the details, for example, paragraph number, and number of characters, for all paragraphs in the `resume` `TextArea`:

```
ObservableList<CharSequence> list = resume.getParagraphs();
int size = list.size();
System.out.println("Paragraph Count:" + size);
for(int i = 0; i < size; i++) {
    CharSequence cs = list.get(i);
    System.out.println("Paragraph #" + (i + 1) +
", Characters=" + cs.length());
```

```

        System.out.println(cs);
    }
}

```

The program in Listing 12-28 shows how to use `TextArea`. It displays a window with a button to print the details of the text in the `TextArea`.

### ***Listing 12-28.*** Using `TextArea` Controls

```

// TextAreaTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class TextAreaTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        TextField title = new TextField("Luci");
        title.setPromptText("Your poem title goes here");

        TextArea poem = new TextArea();
        poem.setPromptText("Your poem goes here");
        poem.setPrefColumnCount(20);
        poem.setPrefRowCount(10);
        poem.appendText("I told her this: her laughter\n" +
                "Is ringing in my ears:\n" +
                "And when I think upon that night\n" +
                "My eyes are dim with tears.");

        Button printBtn = new Button("Print Poem Details");
        printBtn.setOnAction(e -> print(poem));

        VBox root = new VBox(new Label("Title:"), title,
                            new Label("Poem:"), poem, printBtn);
        root.setStyle("-fx-padding: 10;" +
                     "-fx-border-style: solid inside;" +
                     "-fx-border-width: 2;" +
                     "-fx-border-insets: 5;" +
                     "-fx-border-radius: 5;" +
                     "-fx-border-color: blue;");
    }

    Scene scene = new Scene(root);
}

```

```

        stage.setScene(scene);
        stage.setTitle("Using TextArea Controls");
        stage.show();
    }

    public void print(TextArea poem) {
        System.out.println("Poem Length: "
+ poem.getLength());
        System.out.println("Poem Text:\n" + poem.getText());
        System.out.println();

        ObservableList<CharSequence> list
= poem.getParagraphs();
        int size = list.size();
        System.out.println("Paragraph Count:" + size);
        for(int i = 0; i < size; i++) {
            CharSequence cs = list.get(i);
            System.out.println("Paragraph #" + (i + 1) +
", Characters=" + 
cs.length());
            System.out.println(cs);
        }
    }
}

```

## Styling *TextArea* with CSS

The default CSS style-class name for a *TextArea* is `text-area`. It does not add any CSS properties to the ones present in its ancestor `TextInputControl`. It contains `scroll-pane` and `content` substructures, which are a `ScrollPane` and a `Region`, respectively. The `scroll-pane` is the scroll pane that appears when its text exceeds its width or height. The `content` is the region that displays the text.

The following styles set the horizontal and vertical scrollbar policies to `always`, so the scrollbars should always appear in *TextArea*.

Padding for the content area is set to `10px`:

```

.text-area > .scroll-pane {
    -fx-hbar-policy: always;
    -fx-vbar-policy: always;
}

.text-area .content {
    -fx-padding: 10;
}

```

**Tip** At the time of this writing, setting the scrollbar policy for the `scroll-pane` substructure is ignored by the *TextArea*.

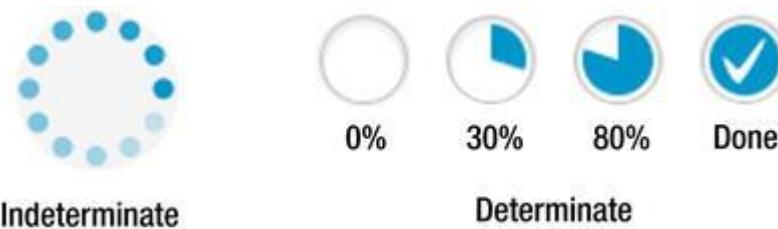
## Showing the Progress of a Task

When you have a long running task, you need to provide a visual feedback to the user about the progress of the task for a better user experience. JavaFX offers two controls to show the progress:

- ProgressIndicator
- ProgressBar

They differ in the ways they display the progress.

The `ProgressBar` class inherits from the `ProgressIndicator` class. `ProgressIndicator` displays the progress in a circular control, whereas `ProgressBar` uses a horizontal bar. The `ProgressBar` class does not add any properties or methods. It just uses a different shape for the control. Figure 12-42 shows a `ProgressIndicator` in indeterminate and determinate states. Figure 12-43 shows a `ProgressBar` in indeterminate and determinate states. Both figures use the same progress values in the four instances of the determinate states.



**Figure 12-42.** A `ProgressIndicator` control in indeterminate and determinate states



**Figure 12-43.** A `ProgressBar` control in indeterminate and determinate states

The current progress of a task may be determined or not. If the progress cannot be determined, it is said to be in an indeterminate state. If the progress is known, it is said to be in a determinate state.

The `ProgressIndicator` class declares two properties:

- `indeterminate`
- `progress`

The `indeterminate` property is a read-only boolean property. If it returns `true`, it means it is not possible to determine the progress. A `ProgressIndicator` in this state is rendered with some kind of

repeated animation. The progress property is a double property. Its value indicates the progress between 0% and 100%. A negative value indicates that the progress is indeterminate. A value between 0 and 1.0 indicates a determinate state with a progress between 0% and 100%. A value greater than 1.0 is treated as 1.0 (i.e., 100% progress).

Both classes provide default constructors that create controls in indeterminate state, as shown in the following code:

```
// Create an indeterminate progress indicator and a progress bar
ProgressIndicator indeterminateInd = new ProgressIndicator();
ProgressBar indeterminateBar = new ProgressBar();
```

The other constructors that take the progress value create controls in the indeterminate or determinate state. If the progress value is negative, they create controls in indeterminate state. Otherwise, they create controls in determinate state, as shown in the following code:

```
// Create a determinate progress indicator with 10% progress
ProgressIndicator indeterminateInd = new ProgressIndicator(0.10);

// Create a determinate progress bar with 70% progress
ProgressBar indeterminateBar = new ProgressBar(0.70);
```

The program in Listing 12-29 shows how to use ProgressIndicator and ProgressBar controls. Clicking the Make Progress button increases the progress by 10%. Clicking the Complete Task button completes the indeterminate tasks by setting their progress to 100%. Typically, the progress properties of these controls are updated by a long running task when the task progresses to a milestone. You used a button to update the progress property to keep the program logic simple.

### ***Listing 12-29.*** Using the ProgressIndicator and ProgressBar Controls

```
// ProgressTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.ProgressBar;
import javafx.scene.control.ProgressIndicator;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class ProgressTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }
}
```

```

@Override
public void start(Stage stage) {
    ProgressIndicator indeterminateInd = new
ProgressIndicator();
    ProgressIndicator determinateInd = new
ProgressIndicator(0);

    ProgressBar indeterminateBar = new ProgressBar();
    ProgressBar determinateBar = new ProgressBar(0);

    Button completeIndBtn = new Button("Complete Task");
    completeIndBtn.setOnAction(e ->
indeterminateInd.setProgress(1.0));

    Button completeBarBtn = new Button("Complete Task");
    completeBarBtn.setOnAction(e ->
determinateBar.setProgress(1.0));

    Button makeProgresstIndBtn = new Button("Make
Progress");
    makeProgresstIndBtn.setOnAction(e ->
makeProgress(determinateInd));

    Button makeProgresstBarBtn = new Button("Make
Progress");
    makeProgresstBarBtn.setOnAction(e ->
makeProgress(determinateBar));

    GridPane root = new GridPane();
    root.setHgap(10);
    root.setVgap(5);
    root.addRow(0, new Label("Indeterminate Progress:"), 
                indeterminateInd, completeIndBtn);
    root.addRow(1, new Label("Determinate Progress:"), 
                determinateInd, makeProgresstIndBtn);
    root.addRow(2, new Label("Indeterminate Progress:"), 
                indeterminateBar, completeBarBtn);
    root.addRow(3, new Label("Determinate Progress:"), 
                determinateBar, makeProgresstBarBtn);
    root.setStyle("-fx-padding: 10;" +
                  "-fx-border-style: solid inside;" +
                  "-fx-border-width: 2;" +
                  "-fx-border-insets: 5;" +
                  "-fx-border-radius: 5;" +
                  "-fx-border-color: blue;");

    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("Using ProgressIndicator and
ProgressBar Controls");
    stage.show();
}

public void makeProgress(ProgressIndicator p) {
    double progress = p.getProgress();
}

```

```
        if (progress <= 0) {
            progress = 0.1;
        } else {
            progress = progress + 0.1;
            if (progress >= 1.0) {
                progress = 1.0;
            }
        }
        p.setProgress(progress);
    }
}
```

## Styling *ProgressIndicator* with CSS

The default CSS style-class name for a ProgressIndicator is `progress-indicator`. ProgressIndicator supports determinate and indeterminate CSS pseudo-classes. The determinate pseudo-class applies when the `indeterminate` property is false. The indeterminate pseudo-class applies when the `indeterminate` property is true.

ProgressIndicator has a CSS style property named `-fx-progress-color`, which is the color of the progress. The following styles set the progress color to red for the indeterminate progress and blue for determinate progress:

```
.progress-indicator:indeterminate {
    -fx-progress-color: red;
}

.progress-indicator:determinate {
    -fx-progress-color: blue;
}
```

The ProgressIndicator contains four substructures:

- An indicator substructure, which is a StackPane
- A progress substructure, which is StackPane
- A percentage substructure, which is a Text
- A tick substructure, which is a StackPane

You can style all substructures of a ProgressIndicator. Please refer to the `modena.css` file for sample code.

## Styling *ProgressIndicator* and Bar with CSS

The default CSS style-class name for a ProgressBar is `progress-bar`. It supports the CSS style properties:

- `-fx-indeterminate-bar-length`

- `-fx-indeterminate-bar-escape`
- `-fx-indeterminate-bar-flip`
- `-fx-indeterminate-bar-animation-time`

All properties apply to the bar that shows the indeterminate progress. The default bar length is 60px. Use the `-fx-indeterminate-bar-length` property to specify a different bar length.

When the `-fx-indeterminate-bar-escape` property is true, the bar starting edge starts at the starting edge of the track and the bar trailing edge ends at the ending edge of the track. That is, the bar is displayed beyond the track length. When this property is false, the bar moves within the track length. The default value is true.

The `-fx-indeterminate-bar-flip` property indicates whether the bar moves only in one direction or both. The default value is true, which means the bar moves in both directions by flipping its direction at the end of each edge.

The `-fx-indeterminate-bar-animation-time` property is the time in seconds that the bar should take to go from one edge to the other. The default value is 2.

The `ProgressBar` contains two substructures:

- A track substructure, which is a `StackPane`
- A bar substructure, which is a `region`

The following styles modify the background color and radius of the bar and track of `ProgressBarControl` to give it a look as shown in Figure 12-44:

```
.progress-bar .track {
    -fx-background-color: lightgray;
    -fx-background-radius: 5;
}

.progress-bar .bar {
    -fx-background-color: blue;
    -fx-background-radius: 5;
}
```



**Figure 12-44.** Customizing the bar and track of the `ProgressBar` control

## Understanding the `TitledPane` Control

`TitledPane` is a labeled control. The `TitledPane` class inherits from the `Labeled` class. A labeled control can have text and a graphic, so it

can have a `TitledPane`. `TitledPane` displays the text as its title. The graphic is shown in the title bar.

Besides text and a graphic, a `TitledPane` has content, which is a `Node`. Typically, a group of controls is placed in a container and the container is added as the content for the `TitledPane`. `TitledPanecan` be in a collapsed or expanded state. In the collapsed state, it displays only the title bar and hides the content. In the expanded state, it displays the title bar and the content. In its title bar, it displays an arrow that indicates whether it is expanded or collapsed. Clicking anywhere in the title bar expands or collapses the content. Figure 12-45 shows a `TitledPane` in both states along with all of its parts.



**Figure 12-45.** A `TitledPane` in the collapsed and expanded states

Use the default constructor to create a `TitledPane` without a title and content. You can set them later using the `setText()` and `setContent()` methods. Alternatively, you can provide the title and content as arguments to its constructor, using the following code:

```
// Create a TitledPane and set its title and content
TitledPane infoPanel = new TitledPane();
infoPanel.setText("Personal Info");
infoPanel.setContent(new Label("Here goes the content."));

// Create a TitledPane with a title and content
TitledPane infoPanel2 = new TitledPane("Personal Info", new
Label("Content"));
```

You can add a graphic to a `TitledPane` using the `setGraphic()` method, which is declared in the `Labeled` class, as shown in the following code:

```
String imageStr = "resources/picture/privacy_icon.png";
URL imageUrl = getClass().getClassLoader().getResource(imageStr);
Image img = new Image(imageUrl.toExternalForm());
ImageView imgView = new ImageView(img);
infoPanel2.setGraphic(imgView);
```

The `TitledPane` class declares four properties:

- animated
- collapsible
- content
- expanded

The `animated` property is a boolean property that indicates whether collapse and expand actions are animated. By default, it is true and those actions are animated. The `collapsible` property is a boolean property that indicates whether the `TitledPane` can collapse. By default, it is set to true and the `TitledPane` can collapse. If you do not want your `TitledPane` to collapse, set this property to false. A noncollapsible `TitledPane` does not display an arrow in its title bar. The `content` property is an `Object` property that stores the reference of any node. The content is visible when the control is in the expanded state. The `expanded` property is a boolean property.

The `TitledPane` is in an expanded state when the property is true. Otherwise, it is in a collapsed state. By default, a `TitledPane` is in an expanded state. Use the `setExpanded()` method to expand and collapse the `TitledPane` programmatically, as shown in the following code:

```
// Set the state to expanded  
infoPane2.setExpanded(true);
```

**Tip** Add a `ChangeListener` to its `expanded` property if you are interested in processing the expanded and collapsed events for a `TitledPane`.

Typically, `TitledPane` controls are used in a group in an `Accordion` control, which displays only one `TitledPane` from the group in the expanded state at a time to save space. You can also use a standalone `TitledPane` if you want to show controls in groups.

**Tip** Recall that the height of a `TitledPane` changes as it expands and collapses. Do not set its minimum, preferred, and maximum heights in your code. Otherwise, it may result in an unspecified behavior.

The program in Listing 12-30 shows how to use the `TitledPane` control. It displays a window with a `TitledPane`, which lets the user enter the first name, last name, and birth date of a person.

### ***Listing 12-30.*** Using the `TitledPane` Control

```
// TitledPaneTest.java  
package com.jdojo.control;
```

```

import java.net.URL;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.DatePicker;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.control.TitledPane;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class TitledPaneTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        TextField firstNameFld = new TextField();
        firstNameFld.setPrefColumnCount(8);

        TextField lastNameFld = new TextField();
        lastNameFld.setPrefColumnCount(8);

        DatePicker dob = new DatePicker();
        dob.setPrefWidth(150);

        GridPane grid = new GridPane();
        grid.addRow(0, new Label("First Name:"), firstNameFld);
        grid.addRow(1, new Label("Last Name:"), lastNameFld);
        grid.addRow(2, new Label("DOB:"), dob);

        TitledPane infoPane = new TitledPane();
        infoPane.setText("Personal Info");
        infoPane.setContent(grid);

        String imageStr
= "resources/picture/privacy_icon.png";
        URL imageUrl
= getClass().getClassLoader().getResource(imageStr);
        Image img = new Image(imageUrl.toExternalForm());
        ImageView imgView = new ImageView(img);
        infoPane.setGraphic(imgView);

        HBox root = new HBox(infoPane);
        root.setSpacing(10);
        root.setStyle("-fx-padding: 10;" +
                     "-fx-border-style: solid inside;" +
                     "-fx-border-width: 2;" +
                     "-fx-border-insets: 5;" +
                     "-fx-border-radius: 5;" +

```

```

        "-fx-border-color: blue;");

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("Using TitledPane Controls");
stage.show();
}

}

```

## Styling *TitledPane* with CSS

The default CSS style-class name for a *TitledPane* is `titled-pane`. *TitledPane* adds two style properties of boolean type:

- `-fx-animated`
- `-fx-collapsible`

The default values for both properties are true. The `-fx-animated` property indicates whether the expanding and collapsing actions are animated. The `-fx-collapsible` property indicates whether the control can be collapsed.

*TitledPane* supports two CSS pseudo-classes:

- `collapsed`
- `expanded`

The `collapsed` pseudo-class applies when the control is collapsed and the `expanded` pseudo-class applies when it is expanded.

*TitledPane* contains two substructures:

- `title`
- `Content`

The `title` substructure is a `StackPane` that contains the content of the title bar. The `titlesubstructure` contains text and arrow-button substructures. The text substructure is a `Label` and it holds the title text and the graphic. The arrow-button substructure is a `StackPane` that contains an arrow substructure, which is also a `StackPane`. The arrow substructure is an indicator that shows whether the control is in an expanded or collapsed state. The content substructure is a `StackPane` that contains the content of the control.

Let's look at an example of the effects of applying the four different styles to a *TitledPane* control, as presented in the following code:

```
/* #1 */
.titled-pane > .title {
```

```

        -fx-background-color: lightgray;
        -fx-alignment: center-right;
    }

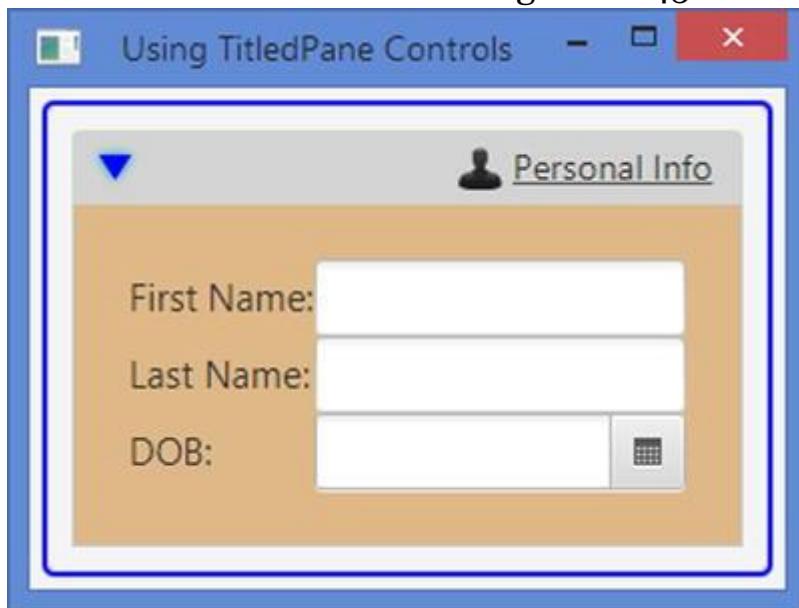
/* #2 */
.titled-pane > .title > .text {
    -fx-font-size: 14px;
    -fx-underline: true;
}

/* #3 */
.titled-pane > .title > .arrow-button > .arrow {
    -fx-background-color: blue;
}

/* #4 */
.titled-pane > .content {
    -fx-background-color: burlywood;
    -fx-padding: 10;
}

```

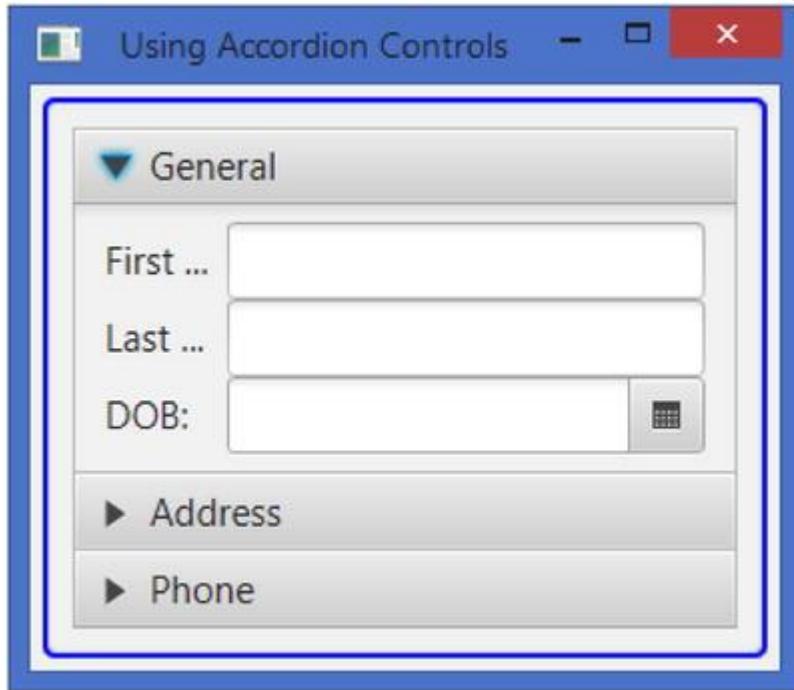
Style #1 sets the background color of the title to light gray and places the graphic and title at the center right in the title bar. Style #2 changes the font size of the title text to 14px and underlines it. Setting the text color of the title using the `-fx-text-fill` property does not work at the time of this writing and setting the `-fx-text-fill` property on the `TitledPane` itself affects the text color of the content as well. Style #3 sets the background color of the arrow to blue. Style #4 sets the background color and padding of the content region. Figure 12-46 shows the same window as shown in Figure 12-45 after applying the above styles.



**Figure 12-46.** Effects of applying styles to a `TitledPane`

## Understanding the Accordion Control

Accordion is a simple control. It displays a group of TitledPane controls where only one of them is in the expanded state at a time. Figure 12-47 shows a window with an Accordion, which contains three TitledPanes. The General TitledPane is expanded. The Address and Phone TitledPanes are collapsed.



**Figure 12-47.** An Accordion with three TitledPanes

The Accordion class contains only one constructor (a no-args constructor) to create its object:

```
// Create an Accordion
Accordion root = new Accordion();
```

Accordion stores the list of its TitledPane controls in an ObservableList<TitledPane>. The getPanels() method returns the list of the TitledPane. Use the list to add or remove any TitledPane to the Accordion, as shown in the following code:

```
TitledPane generalPane = new TitledPane();
TitledPane addressPane = new TitledPane();
TitledPane phonePane = new TitledPane();
...
Accordion root = new Accordion();
root.getPanels().addAll(generalPane, addressPane, phonePane);
```

The Accordion class contains an expandedPane property, which stores the reference of the currently expanded TitledPane. By default, an Accordion displays all of its TitledPanes in a collapsed state, and this property is set to null. Click the title bar of a TitledPane or use the setExpandedPane() method to expand a TitledPane. Add

a ChangeListener to this property if you are interested in when the expanded TitledPane changes. The program in Listing 12-31 shows how to create and populate an Accordion.

### ***Listing 12-31.*** Using the TitledPane Control

```
// AccordionTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Accordion;
import javafx.scene.control.DatePicker;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.control.TitledPane;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class AccordionTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        TitledPane generalPane = this.getGeneralPane();
        TitledPane addressPane = this.getAddressPane();
        TitledPane phonePane = this.getPhonePane();

        Accordion root = new Accordion();
        root.getPanes().addAll(generalPane, addressPane,
        phonePane);
        root.setExpandedPane(generalPane);
        root.setStyle("-fx-padding: 10;" +
                     "-fx-border-style: solid inside;" +
                     "-fx-border-width: 2;" +
                     "-fx-border-insets: 5;" +
                     "-fx-border-radius: 5;" +
                     "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using Accordion Controls");
        stage.show();
    }

    public TitledPane getGeneralPane() {
        GridPane grid = new GridPane();
        grid.addRow(0, new Label("First Name:"), new
TextField());
        grid.addRow(1, new Label("Last Name:"), new
TextField());
    }
}
```

```

        grid.addRow(2, new Label("DOB:"), new DatePicker()));

        TitledPane generalPane = new TitledPane("General",
grid);
        return generalPane;
    }

    public TitledPane getAddressPane() {
        GridPane grid = new GridPane();
        grid.addRow(0, new Label("Street:"), new
TextField());
        grid.addRow(1, new Label("City:"), new TextField());
        grid.addRow(2, new Label("State:"), new
TextField());
        grid.addRow(3, new Label("ZIP:"), new TextField());

        TitledPane addressPane = new TitledPane("Address",
grid);
        return addressPane;
    }

    public TitledPane getPhonePane() {
        GridPane grid = new GridPane();
        grid.addRow(0, new Label("Home:"), new TextField());
        grid.addRow(1, new Label("Work:"), new TextField());
        grid.addRow(2, new Label("Cell:"), new TextField());

        TitledPane phonePane = new TitledPane("Phone",
grid);
        return phonePane;
    }
}

```

## Styling Accordion with CSS

The default CSS style-class name for an Accordion is `accordion`. Accordion does not add any CSS properties. It contains a first-titled-pane substructure, which is the first `TitledPane`. The following style sets the background color and insets of the title bar of all `TitledPanes`:

```

.accordion > .titled-pane > .title {
    -fx-background-color: burlywood;
    -fx-background-insets: 1;
}

```

The following style sets the background color of the title bar of the first `TitledPane` of the Accordion:

```

.accordion > .first-titled-pane > .title {
    -fx-background-color: derive(red, 80%);
}

```

## Understanding the *Pagination Control*

Pagination is used to display a large single content by dividing sections of it into smaller chunks called pages, for example, the results of a search. Figure 12-48 shows a Pagination control.

A Pagination control has a page count, which is the number of pages in it. If the number of pages is not known, the page count may be indeterminate. Each page has an index, which starts at 0.

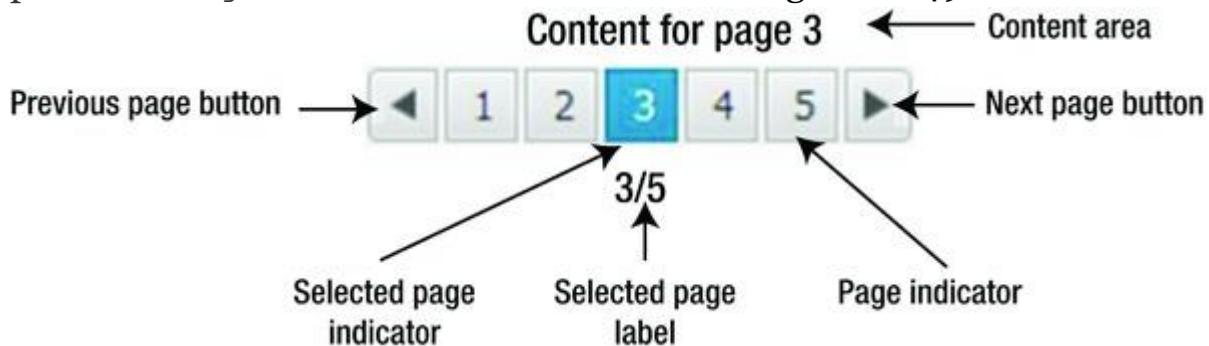


**Figure 12-48.** A Pagination control

A Pagination control is divided into two areas:

- Content area
- Navigation area

The content area displays the content of the current page. The navigation area contains parts to allow the user to navigate from one page to another. You can navigate between pages sequentially or randomly. The parts of a Pagination control are shown in Figure 12-49.



**Figure 12-49.** Parts of a Pagination control

The previous and next page arrow buttons let the user navigate to the previous and next pages, respectively. The previous page button is disabled when you are on the first page. The next page button is disabled when you are on the last page. Page indicators also let you navigate to a specific page by showing all of the page numbers. By default, page indicators use a tool tip to show the page number, which you have the option to disable using a CSS property. The selected page indicator shows the current page. The selected page label shows the current page selection details.

The `Pagination` class provides several constructors. They configure the control differently. The default constructor creates a control with an indeterminate page count and zero as the index for the selected page, as in the following code:

```
// Indeterminate page count and first page selected  
Pagination pagination1 = new Pagination();
```

When the page count is indeterminate, the page indicator label displays `x/...`, where `x` is the current page index plus 1.

You use another constructor to specify a page count, as in the following code:

```
// 5 as the page count and first page selected  
Pagination pagination2 = new Pagination(5);
```

You can use yet another constructor to specify the page count and the selected page index, as in the following code:

```
// 5 as the page count and second page selected (page index  
starts at 0)
```

```
Pagination pagination3 = new Pagination(5, 1);
```

The `Pagination` class declares an `INDETERMINATE` constant that can be used to specify an indeterminate page count, as in the following code:

```
// Indeterminate page count and second page selected  
Pagination pagination4 = new Pagination(Pagination.INDETERMINATE,  
1);
```

The `Pagination` class contains the following properties:

- `currentPageIndex`
- `maxPageIndicatorCount`
- `pageCount`
- `pageFactory`

The `currentPageIndex` is an integer property. Its value is the page index of the page to display. The default value is zero. You can specify its value using one of the constructors or using

the `setCurrentPageIndex()` method. If you set its value to less than zero, the first page index, which is zero, is set as its value. If you set its value to greater than the page count minus 1, its value is set to page count minus 1. If you want to know when a new page is displayed, add a `ChangeListener` to the `currentPageIndex` property.

The `maxPageIndicatorCount` is an integer property. It sets the maximum number of page indicators to display. It defaults to 10. Its value remains unchanged if it is set beyond the page count range. If its value is set too high, the value is reduced so that the number of page indicators fits the control. You can set its value using the `setMaxPageIndicatorCount()` method.

The `pageCount` is an integer property. It is the number of pages in the `Pagination` control. Its value must be greater than or equal to 1. It defaults to indeterminate. Its value can be set in the constructors or using the `setPageCount()` method.

The `pageFactory` is the most important property. It is an object property of the `Callback<Integer, Node>` type. It is used to generate pages. When a page needs to be displayed, the control calls the `call()` method of the `Callback` object passing the page index. The `call()` method returns a node that is the content of the page. The following snippet of code creates and sets a page factory for a `Pagination` control. The page factory returns a `Label`:

```
// Create a Pagination with an indeterminate page count
Pagination pagination = new Pagination();

// Create a page factory that returns a Label
Callback<Integer, Node> factory = pageIndex -> new Label("Content
for page " + (pageIndex + 1));

// Set the page factory
pagination.setPageFactory(factory);
```

**Tip** The `call()` method of the page factory should return `null` if a page index does not exist. The current page does not change when the `call()` method returns `null`.

The program in Listing 12-32 shows how to use a `Pagination` control. It sets the page count to 5. The page factory returns a `Label` with text that shows the page number. It will display a window with a `Pagination` control similar to the one shown in Figure 12-48.

### ***Listing 12-32.*** Using the `Pagination` Control

```
// PaginationTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.Pagination;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class PaginationTest extends Application {
    private static final int PAGE_COUNT = 5;

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

    @Override
```

```

public void start(Stage stage) {
    Pagination pagination = new Pagination(PAGE_COUNT);

    // Set the page factory
    pagination.setPageFactory(this::getPage);

    VBox root = new VBox(pagination);
    root.setStyle("-fx-padding: 10;" +
                  "-fx-border-style: solid inside;" +
                  "-fx-border-width: 2;" +
                  "-fx-border-insets: 5;" +
                  "-fx-border-radius: 5;" +
                  "-fx-border-color: blue;");

    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("Using Pagination Controls");
    stage.show();
}

public Label getPage(int pageIndex) {
    Label content = null;

    if (pageIndex >= 0 && pageIndex < PAGE_COUNT) {
        content = new Label("Content for page " +
+ (pageIndex + 1));
    }
    return content;
}
}

```

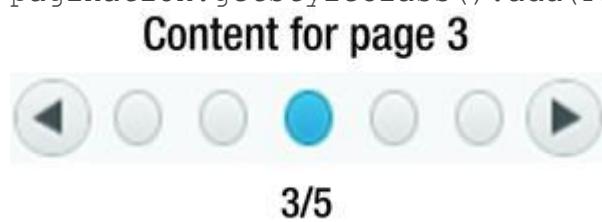
The page indicators may be numeric buttons or bullet buttons. Numeric buttons are used by default. The `Pagination` class contains a `String` constant named `STYLE_CLASS_BULLET`, which is the style class for the control if you want to use bullet buttons. The following snippet of code creates a `Pagination` control and sets its style class to use bullet buttons as page indicators. Figure 12-50 shows a `Pagination` control with bullet buttons as page indicators.

```

Pagination pagination = new Pagination(5);

// Use bullet page indicators
pagination.getStyleClass().add(Pagination.STYLE_CLASS_BULLET);

```



**Figure 12-50.** A `Pagination` control using bullet buttons as page indicators

## Styling `Pagination` with CSS

The default CSS style-class name for a **Pagination** control is `pagination`. **Pagination** adds several CSS properties:

- `-fx-max-page-indicator-count`
- `-fx-arrows-visible`
- `-fx-tooltip-visible`
- `-fx-page-information-visible`
- `-fx-page-information-alignment`

The `-fx-max-page-indicator-count` property specifies the maximum number of page indicators to display. The default value is 10. The `-fx-arrows-visible` property specifies whether the previous and next page buttons are visible. The default value is true. The `-fx-tooltip-visible` property specifies whether a tool tip is displayed when the mouse hovers over a page indicator. The default value is true. The `-fx-page-information-visible` specifies whether the selected page label is visible. The default value is true. The `-fx-page-information-alignment` specifies the location of the selected page label relative to the page indicators. The possible values are top, right, bottom, and left. The default value is bottom, which displays the selected page indicator below the page indicators.

The **Pagination** control has two substructures of `StackPane` type:

- `page`
- `pagination-control`

The `page` substructure represents the content area. The `pagination-control` substructure represents the navigation area and it has the following substructures:

- `left-arrow-button`
- `right-arrow-Button`
- `bullet-button`
- `number-button`
- `page-information`

The `left-arrow-button` and `right-arrow-button` substructures are of the `Button` type. They represent the previous and next page buttons, respectively. The `left-arrow-button` substructure has a `left-arrow` substructure, which is a `StackPane`, and it represents the arrow in the previous page button. The `right-arrow-button` substructure has a `right-arrow` substructure, which is a `StackPane`, and it represents the arrow in the next page button.

The bullet-button and number-button are of the `ToggleButton` type, and they represent the page indicators. The page-information substructure is a `Label` that holds the selected page information. The pagination-control substructure holds the previous and next page buttons and the page indicators in a substructure called `control-box`, which is an `HBox`.

The following styles make the selected page label invisible, set the page background to light gray, and draw a border around the previous, next, and page indicator buttons. Please refer to the `modena.css` file for more details on how to style a Pagination control.

```
.pagination {
    -fx-page-information-visible: false;
}

.pagination > .page {
    -fx-background-color: lightgray;
}

.pagination > .pagination-control > .control-box {
    -fx-padding: 2;
    -fx-border-style: dashed;
    -fx-border-width: 1;
    -fx-border-radius: 5;
    -fx-border-color: blue;
}
```

## Understanding the Tool Tip Control

A tool tip is a pop-up control used to show additional information about a node. It is displayed when a mouse pointer hovers over the node. There is a small delay between when the mouse pointer hovers over a node and when the tool tip for the node is shown. The tool tip is hidden after a small period. It is also hidden when the mouse pointer leaves the control. You should not design a GUI application where the user depends on seeing tool tips for controls, as they may not be shown at all if the mouse pointer never hovers over the controls. Figure 12-51 shows a window with a tool tip, which displays Saves the data text.



**Figure 12-51.** A window showing a tool tip

A tool tip is represented by an instance of the `Tooltip` class, which inherits from the `PopupControl` class. A tool tip can have text and a graphic. You can create a tool tip using its default constructor, which has no text and no graphic. You can also create a tool tip with text using the other constructor, as in the following code:

```
// Create a Tooltip with No text and no graphic
Tooltip tooltip1 = new Tooltip();
```

```
// Create a Tooltip with text
Tooltip tooltip2 = new Tooltip("Closes the window");
```

A tool tip needs to be installed for a node using the `install()` static method of the `Tooltip` class. Use the `uninstall()` static method to uninstall a tool tip for a node:

```
Button saveBtn = new Button("Save");
Tooltip tooltip = new Tooltip("Saves the data");
```

```
// Install a tooltip
Tooltip.install(saveBtn, tooltip);
```

```
...
// Uninstall the tooltip
Tooltip.uninstall(saveBtn, tooltip);
```

Tool tips are frequently used for UI controls. Therefore, installing tool tips for controls has been made easier. The `Control` class contains a `tooltip` property, which is an object property of the `Tooltip` type. You can use the `setTooltip()` method of the `Control` class to set a `Tooltip` for controls. If a node is not a control, for example, a `Circle` node, you will need to use the `install()` method to set a tool tip as shown above. The following snippet of code shows how to use the `tooltip` property for a button:

```
Button saveBtn = new Button("Save");
```

```
// Install a tooltip
saveBtn.setTooltip(new Tooltip("Saves the data"));
```

```
...
// Uninstall the tooltip
saveBtn.setTooltip(null);
```

**Tip** A tool tip can be shared among multiple nodes. A tool tip uses a `Label` control to display its text and graphic. Internally, all content-related properties set on a tool tip are delegated to the `Label` control.

The `Tooltip` class contains several properties:

- `text`
- `graphic`
- `contentDisplay`
- `textAlignment`
- `textOverrun`

- wrapText
- graphicTextGap
- font
- activated

The `text` property is a `String` property, which is the text to be displayed in the tool tip. The `graphic` property is an object property of the `Node` type. It is an icon for the tool tip.

The `contentDisplay` property is an object property of the `ContentDisplay` enum type. It specifies the position of the graphic relative to the text. The possible value is one of the constants in the `ContentDisplay` enum: `TOP`, `RIGHT`, `BOTTOM`, `LEFT`, `CENTER`, `TEXT_ONLY`, and `GRAPHIC_ONLY`. The default value is `LEFT`, which places the graphic left to the text.

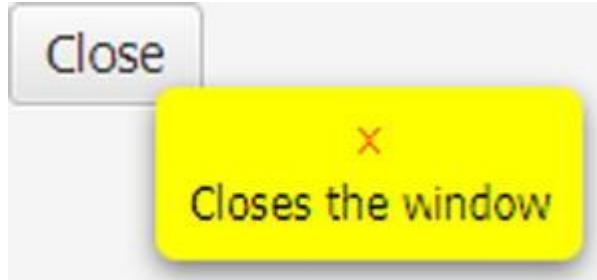
The following snippet of code uses an icon for a tool tip and places it above the text. The icon is just a `Label` with X as its text. Figure 12-52 shows how the tool tip looks.

```
// Create and configure the Tooltip
Tooltip closeBtnTip = new Tooltip("Closes the window");
closeBtnTip.setStyle("-fx-background-color: yellow; -fx-text-
fill: black;");

// Display the icon above the text
closeBtnTip.setContentDisplay(ContentDisplay.TOP);

Label closeTipIcon = new Label("X");
closeTipIcon.setStyle("-fx-text-fill: red;");
closeBtnTip.setGraphic(closeTipIcon);
```

```
// Create a Button and set its Tooltip
Button closeBtn = new Button("Close");
closeBtn.setTooltip(closeBtnTip);
```



**Figure 12-52.** Using an icon and placing it at the top of the text in a tool tip

The `textAlignment` property is an object property of the `TextAlignment` enum type. If specifies the text alignment when the text spans multiple lines. The possible value is one of the constants in the `TextAlignment` enum: `LEFT`, `RIGHT`, `CENTER`, and `JUSTIFY`.

The `textOverrun` property is an object property of the `OverrunStyle` enum type. It specifies the behavior to use when there is not enough space in the tool tip to display the entire text. The default behavior is to use an ellipsis.

The `wrapText` is a boolean property. It specifies whether text should be wrapped onto another line if its run exceeds the width of the tool tip. The default value is false.

The `graphicTextGap` property is a double property that specifies the space between the text and graphic in pixel. The default value is 4.

The `font` property is an object property of the `Font` type. It specifies the default font to use for the text. The `activated` property is a read-only boolean property. It is true when the tool tip is activated.

Otherwise, it is false. A tool tip is activated when the mouse moves over a control, and it is shown after it is activated.

The program in Listing 12-33 shows how to create, configure, and set tool tips for controls. After you run the application, place the mouse pointer over the name field, Save button, and Close button. After a short time, their tool tips will be displayed. The tool tip for the Close button looks different from that of the Save button. It uses an icon and different background and text colors.

### ***Listing 12-33.*** Using the Tooltip Control

```
// TooltipTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.ContentDisplay;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.control.Tooltip;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class TooltipTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Label nameLbl = new Label("Name:");
        TextField nameFld = new TextField();
        Button saveBtn = new Button("Save");
        Button closeBtn = new Button("Close");
        ...
    }
}
```

```

        // Set an ActionEvent handler
        closeBtn.setOnAction(e -> stage.close());

        // Add tooltips for Name field and Save button
        nameFld.setTooltip(new Tooltip("Enter your
name\n(Max. 10 chars)"));
        saveBtn.setTooltip(new Tooltip("Saves the data"));

        // Create and configure the Tooltip for Close button
        Tooltip closeBtnTip = new Tooltip("Closes the
window");
        closeBtnTip.setStyle("-fx-background-color: yellow;
" +
                           " -fx-text-fill: black;");

        // Display the icon above the text
        closeBtnTip.setContentDisplay(ContentDisplay.TOP);

        Label closeTipIcon = new Label("X");
        closeTipIcon.setStyle("-fx-text-fill: red;");
        closeBtnTip.setGraphic(closeTipIcon);

        // Set its Tooltip for Close button
        closeBtn.setTooltip(closeBtnTip);

        HBox root = new HBox(nameLbl, nameFld, saveBtn,
closeBtn);
        root.setSpacing(10);
        root.setStyle("-fx-padding: 10;" +
                     "-fx-border-style: solid inside;" +
                     "-fx-border-width: 2;" +
                     "-fx-border-insets: 5;" +
                     "-fx-border-radius: 5;" +
                     "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using Tooltip Controls");
        stage.show();
    }
}

```

## Styling *Tooltip* with CSS

The default CSS style-class name for a *Tooltip* control is `tooltip`. *Tooltip* add several CSS properties:

- `-fx-text-alignment`
- `-fx-text-overrun`
- `-fx-wrap-text`
- `-fx-graphic`
- `-fx-content-display`
- `-fx-graphic-text-gap`

- `-fx-font`

All of the CSS properties correspond to the content-related properties in the `Tooltip` class. Please refer to the previous section for the description of all these properties. The following code sets the background color, text color, and the wrap text properties for `Tooltip`:

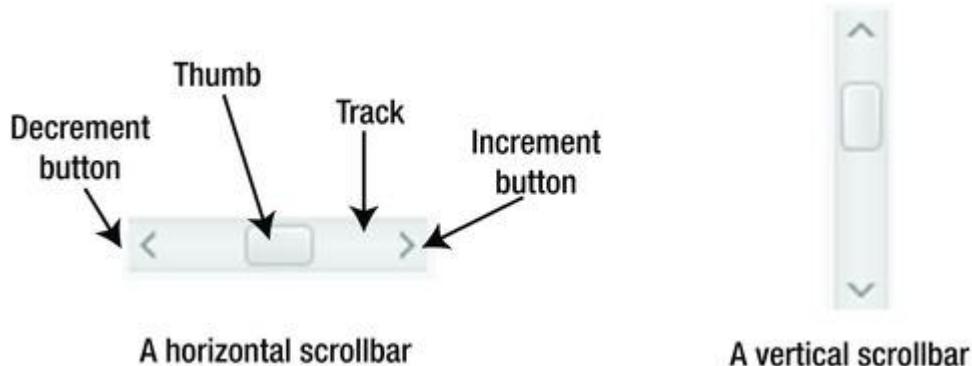
```
.tooltip {
    -fx-background-color: yellow;
    -fx-text-fill: black;
    -fx-wrap-text: true;
}
```

## Providing Scrolling Features in Controls

JavaFX provides two controls named `ScrollBar` and `ScrollPane` that provide scrolling features to other controls. Typically, these controls are not used alone. They are used to support scrolling in other controls.

### Understanding the *ScrollBar* Control

`ScrollBar` is a basic control that does not provide the scrolling feature by itself. It is represented as a horizontal or vertical bar that lets users choose a value from a range of values. Figure 12-53 shows a horizontal and a vertical scrollbar.



**Figure 12-53.** Horizontal and vertical scrollbars with their parts

A `ScrollBar` control consists of four parts:

- An increment button to increase the value
- A decrement button to decrease the value
- A thumb (or knob) to show the current value
- A track where the thumb moves

The increment and decrement buttons in a vertical `ScrollBar` are on the bottom and top, respectively.

The `ScrollBar` class provides a default constructor that creates a horizontal scrollbar. You can set its orientation to vertical using the `setOrientation()` method:

```
// Create a horizontal scroll bar
ScrollBar hsb = new ScrollBar();

// Create a vertical scroll bar
ScrollBar vsb = new ScrollBar();
vsb.setOrientation(Orientation.VERTICAL);
```

The `min` and `max` properties represent the range of its value.

Its `value` property is the current value. The default values for `min`, `max`, and `value` properties are 0, 100, and 0, respectively. If you are interested in knowing when the `value` property changes, you need to add a `ChangeListener` to it. The following code would set the `value` properties to 0, 200, and 150:

```
ScrollBar hsb = new ScrollBar();
hsb.setMin(0);
hsb.setMax(200);
hsb.setValue(150);
```

The current value of a scrollbar may be changed three different ways:

- Programmatically using the `setValue()`, `increment()`, and `decrement()` methods
- By the user dragging the thumb on the track
- By the user clicking the increment and decrement buttons

The `blockIncrement` and `unitIncrement` properties specify the amount to adjust the current value when the user clicks the track and the increment or decrement buttons, respectively. Typically, the block increment is set to a larger value than the unit increment.

The default CSS style-class name for a `ScrollBar` control is `scroll-bar`. `ScrollBar` supports two CSS pseudo-classes: `horizontal` and `vertical`. Some of its properties can be set using CSS.

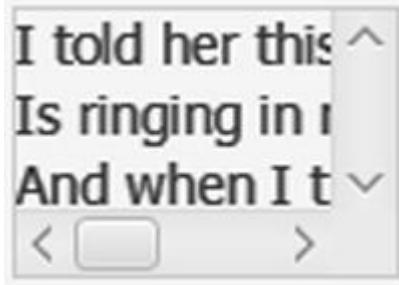
`ScrollBar` is rarely used directly by developers. It is used to build complete controls that support scrolling, for example, the `ScrollPane` control. If you need to provide scrolling capability to a control, use the `ScrollPane`, which I will discuss in the next section.

## Understanding the `ScrollPane` Control

A `ScrollPane` provides a scrollable view of a node.

A `ScrollPane` consists of a horizontal `ScrollBar`, a vertical `ScrollBar`, and a content node. The node for which

the `ScrollPane` provides scrolling is the content node. If you want to provide a scrollable view of multiple nodes, add them to a layout pane, for example, a `GridPane`, and then, add the layout pane to the `ScrollPane` as the content node. `ScrollPane` uses a scroll policy to specify when to show a specific scrollbar. The area through which the content is visible is known as *viewport*. Figure 12-54 shows a `ScrollPane` with a `Label` as its content node.



**Figure 12-54.** A `ScrollPane` with a `Label` as its content node

**Tip** Some of the commonly used controls that need scrolling capability, for example, a `TextArea`, provide a built-in `ScrollPane`, which is part of such controls.

You can use the constructors of the `ScrollPane` class to create an empty `ScrollPane` or a `ScrollPane` with a content node, as shown in the following code. You can set the content node later using the `setContent()` method.

```
Label poemLbl1 = ...
Label poemLbl2 = ...

// Create an empty ScrollPane
ScrollPane sPanel1 = new ScrollPane();

// Set the content node for the ScrollPane
sPanel1.setContent(poemLbl1);

// Create a ScrollPane with a content node
ScrollPane sPanel2 = new ScrollPane(poemLbl2);
```

**Tip** The `ScrollPane` provides the scrolling for its content based on the layout bounds of the content. If the content uses effects or transformation, for example, scaling, you need to wrap the content in a `Group` and add the `Group` to the `ScrollPane` get proper scrolling.

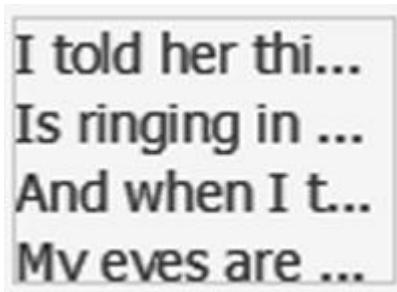
The `ScrollPane` class contains several properties, most of which are commonly not used by developers:

- `content`
- `pannable`

- fitToHeight
- fitToWidth
- hbarPolicy
- vbarPolicy
- hmin
- hmax
- hvalue
- vmin
- vmax
- vvalue
- prefViewportHeight
- prefViewportWidth
- viewportBounds

The content property is an object property of the Node type and it specifies the content node. You can scroll the content using the scrollbars or by panning. If you use panning, you need to drag the mouse while left, right, or both buttons are pressed to scroll the content. By default, a ScrollPane is not pannable and you need to use the scrollbars to scroll through the content. The pannable property is a boolean property that specifies whether the ScrollPane is pannable. Use the setPannable(true) method to make a ScrollPane pannable.

The fitToHeight and fitToWidth properties specify whether the content node is resized to match the height and width of the viewport, respectively. By default, they are false. These properties are ignored if the content node is not resizable. Figure 12-55 shows the same ScrollPane as shown in Figure 12-54 with its fitToHeight and fitToWidth properties set to true. Notice that the Labelcontent node has been resized to fit into the viewport.



**Figure 12-55.** A ScrollPane with fitToHeight and fitToWidth properties set to true

The hbarPolicy and vbarPolicy properties are object properties of the ScrollPane. ScrollBarPolicy enum type. They specify when to

show the horizontal and vertical scrollbars. The possible values are ALWAYS, AS\_NEEDED, and NEVER. When the policy is set to ALWAYS, the scrollbar is shown all the time. When the policy is set to AS\_NEEDED, the scrollbar is shown when required based on the size of the content. When the policy is set to NEVER, the scrollbar is never shown.

The `hmin`, `hmax`, and `hvalue` properties specify the min, max, and value properties of the horizontal scrollbar, respectively. The `vmin`, `vmax`, and `vvalue` properties specify the min, max, and value properties of the vertical scrollbar, respectively. Typically, you do not set these properties. They change based on the content and as the user scrolls through the content.

The `prefViewportHeight` and `prefViewportWidth` are the preferred height and width, respectively, of the viewport that is available to the content node.

The `viewportBounds` is an object property of the `Bounds` type. It is the actual bounds of the viewport. The program in Listing 12-34 shows how to use a `ScrollPane`. It sets a `Label` with four lines of text as its content. It also makes the `ScrollPane` pannable. That is, you can drag the mouse clicking its button to scroll through the text.

### ***Listing 12-34.*** Using `ScrollPane`

```
// ScrollPaneTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.ScrollPane;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class ScrollPaneTest extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage stage) {
        Label poemLbl = new Label("I told her this; her
laughter light\n" +
                               "Is ringing in my ears;\n" +
                               "And when I think upon that
night\n" +
                               "My eyes are dim with tears.");
    }

    // Create a scroll pane with poemLbl as its content
    ScrollPane sPane = new ScrollPane(poemLbl);
}
```

```

sPane.setPannable(true);

HBox root = new HBox(sPane);
root.setStyle("-fx-padding: 10;" +
              "-fx-border-style: solid inside;" +
              "-fx-border-width: 2;" +
              "-fx-border-insets: 5;" +
              "-fx-border-radius: 5;" +
              "-fx-border-color: blue;");

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("Using ScrollPane Controls");
stage.show();
}
}

```

The default CSS style-class name for a `ScrollPane` control is `scroll-pane`. Please refer to the `modena.css` file for sample styles and the online *JavaFX CSS Reference Guide* for the complete list of CSS properties and pseudo-classes supported by the `ScrollPane`.

## Keeping Things Separate

Sometimes you may want to place logically related controls side by side horizontally or vertically. For better appearance, controls are grouped using different types of separators. Sometimes using a border suffices; but sometimes you will use the `TitledPane` controls.

The `Separator` and `SplitPane` controls are solely meant for visually separating two controls or two groups of controls.

### Understanding the *Separator* Control

A `Separator` is a horizontal or vertical line that separates two groups of controls. Typically, they are used in menus or combo boxes. Figure 12-56 shows menu items of a restaurant separated by horizontal and vertical separators.



**Figure 12-56.** Using horizontal and vertical separators

The default constructor creates a horizontal Separator. To create a vertical Separator, you can specify a vertical orientation in the constructor or use the `setOrientation()` method, as shown in the following code:

```
// Create a horizontal separator
Separator separator1 = new Separator();

// Change the orientation to vertical
separator1.setOrientation(Orientation.VERTICAL);

// Create a vertical separator
Separator separator2 = new Separator(Orientation.VERTICAL);
```

A separator resizes itself to fill the space allocated to it. A horizontal Separator resizes horizontally and a vertical Separator resizes vertically. Internally, a Separator is a Region. You can change its color and thickness using a CSS.

The Separator class contains three properties:

- `orientation`
- `halignment`
- `valignment`

The `orientation` property specifies the orientation of the control. The possible values are one of the two constants of the `Orientation` enum: `HORIZONTAL` and `VERTICAL`.

The `halignment` property specifies the horizontal alignment of the separator line within the width of a vertical separator. This property is ignored for a horizontal separator. The possible values are one of the constants of the `HPosenum`: `LEFT`, `CENTER`, and `RIGHT`. The default value is `CENTER`. The `valignment` property specifies the vertical alignment of the separator line within the height of a horizontal separator. This property is ignored for a vertical separator. The possible values are one of the constants of the `VPos` enum: `BASELINE`, `TOP`, `CENTER`, and `BOTTOM`. The default value is `CENTER`.

### Styling Separator with CSS

The default CSS style-class name for a Separator control is `separator`. Separator contains CSS properties, which corresponds to its Java properties:

- `-fx-orientation`

- `-fx-halignment`
- `-fx-valignment`

Separator supports horizontal and vertical CSS pseudo-classes that apply to horizontal and vertical separators, respectively. It contains a line substructure that is a Region. The line you see in a separator is created by specifying the border for the line substructure. The following style was used to create the separators in Figure 12-56:

```
.separator > .line {  
    -fx-border-style: solid;  
    -fx-border-width: 1;  
}
```

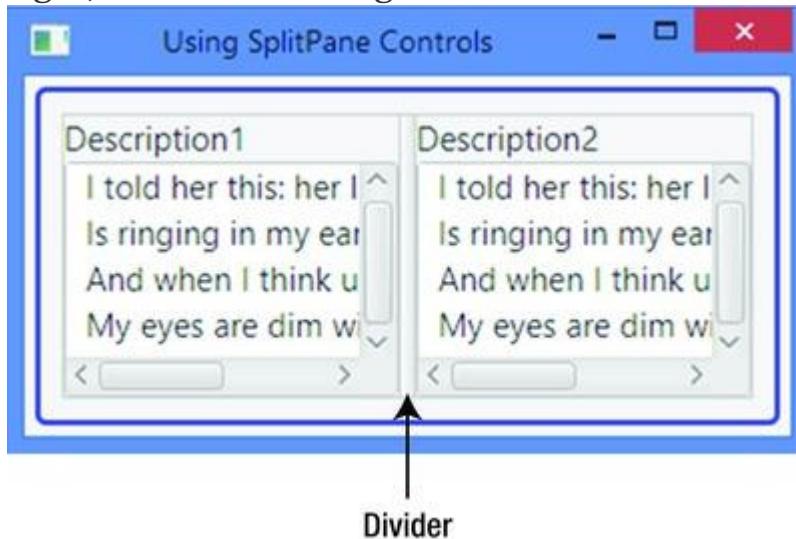
You can use an image as a separator. Set the appropriate width or height of the separator and use an image as the background image. The following code assumes that the `separator.jpg` image file exists in the same directory as the CSS file containing the style. The styles set the preferred height of the horizontal separator and the preferred width of the vertical separator to 10px.

```
.separator {  
    -fx-background-image: url("separator.jpg");  
    -fx-background-repeat: repeat;  
    -fx-background-position: center;  
    -fx-background-size: cover;  
}  
  
.separator:horizontal {  
    -fx-pref-height: 10;  
}  
  
.separator:vertical {  
    -fx-pref-width: 10;  
}
```

## Understanding the *SplitPane* Control

SplitPane arranges multiple nodes by placing them horizontally or vertically separated by a divider. The divider can be dragged by the user, so the node on one side of the divider expands and the node on the other side shrinks by the same amount. Typically, each node in a SplitPane is a layout pane containing some controls. However, you can use any node, for example, a Button. If you have used Windows Explorer, you are already familiar with using a SplitPane. In a Windows Explorer, the divider separates the tree view and the list view. Using the divider, you can resize the width of the tree view and the width of the list view resizes with the equal amount in the opposite direction. A resizable HTML frameset works similar to a SplitPane. Figure 12-57 shows a window with a horizontal SplitPane.

The `SplitPane` contains two `VBox` layout panes, each of them contains a `Label` and a `TextArea`. Figure 12-57 shows the divider dragged to the right, so the left `VBox` gets more width than the right one.



**Figure 12-57.** A window with a horizontal `SplitPane`

You can create a `SplitPane` using the default constructor of the `SplitPane` class:

```
SplitPane sp = new SplitPane();
```

The `getItems()` method of the `SplitPane` class returns the `ObservableList<Node>` that stores the list of nodes in a `SplitPane`. Add all your nodes to this list, as shown in the following code:

```
// Create panes
GridPane leftPane = new GridPane();
GridPane centerPane = new GridPane();
GridPane rightPane = new GridPane();

/* Populate the left, center, and right panes with controls here
 */

// Add panels to the a SplitPane
SplitPane sp = new SplitPane();
sp.getItems().addAll(leftPane, centerPane, rightPane);
```

By default, `SplitPane` places its nodes horizontally.

Its `orientation` property can be used to specify the orientation:

```
// Place nodes vertically
sp.setOrientation(Orientation.VERTICAL);
```

A divider can be moved between the leftmost and rightmost edges or topmost and bottommost edges provided it does not overlap any other divider. The divider position can be set between 0 and 1. The position 0 means topmost or leftmost. The position 1 means bottommost or rightmost. By default, a divider is placed in the middle with its position

set to 0.5. Use either of the following two methods to set the position of a divider:

- `setDividerPositions(double... positions)`
- `setDividerPosition(int dividerIndex, double position)`

The `setDividerPositions()` method takes the positions of multiple dividers. You must provide positions for all dividers from starting up to the one you want to set the positions.

If you want to set the position for a specific divider, use the `setDividerPosition()` method. The first divider has the index 0. Positions passed in for an index outside the range are ignored.

The `getDividerPositions()` method returns the positions of all dividers. It returns a `double` array. The index of dividers matches the index of the array elements.

By default, `SplitPane` resizes its nodes when it is resized. You can prevent a specific node from resizing with the `SplitPane` using the `setResizableWithParent()` static method:

```
// Make node1 non-resizable
SplitPane.setResizableWithParent(node1, false);
```

The program in Listing 12-35 shows how to use `SplitPane`. It displays a window as shown in Figure 12-57. Run the program and use the mouse to drag the divider to the left or right to adjust the spacing for the left and right nodes.

### ***Listing 12-35.*** Using `SplitPane` Controls

```
// SplitPaneTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.SplitPane;
import javafx.scene.control.TextArea;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class SplitPaneTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
```

```

        TextArea desc1 = new TextArea();
        desc1.setPrefColumnCount(10);
        desc1.setPrefRowCount(4);

        TextArea desc2 = new TextArea();
        desc2.setPrefColumnCount(10);
        desc2.setPrefRowCount(4);

        VBox vb1 = new VBox(new Label("Description1"),
desc1);
        VBox vb2 = new VBox(new Label("Description2"),
desc2);

        SplitPane sp = new SplitPane();
        sp.getItems().addAll(vb1, vb2);

        HBox root = new HBox(sp);
        root.setSpacing(10);
        root.setStyle("-fx-padding: 10;" +
                     "-fx-border-style: solid inside;" +
                     "-fx-border-width: 2;" +
                     "-fx-border-insets: 5;" +
                     "-fx-border-radius: 5;" +
                     "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using SplitPane Controls");
        stage.show();
    }
}

```

## Styling *SplitPane* with CSS

The default CSS style-class name for a `SplitPane` control is `split-pane`. `SplitPane` contains `-fx-orientation` CSS properties, which determine its orientation. The possible values are `horizontal` and `vertical`.

`SplitPane` supports horizontal and vertical CSS pseudo-classes that apply to horizontal and vertical `SplitPanes`, respectively. The divider is a `split-pane-divider` substructure of the `SplitPane`, which is a `StackPane`. The following code sets a blue background color for dividers, 5px preferred width for dividers in a horizontal `SplitPane`, and 5px preferred height for dividers in a vertical `SplitPane`:

```

.split-pane > .split-pane-divider {
    -fx-background-color: blue;
}

.split-pane:horizontal > .split-pane-divider {
    -fx-pref-width: 5;
}

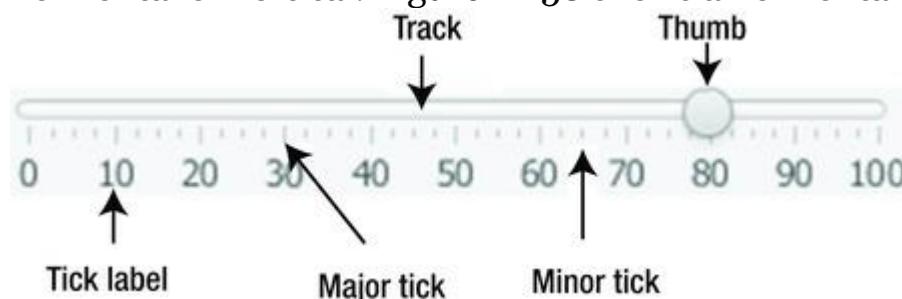
```

```
.split-pane:vertical > .split-pane-divider {
    -fx-pref-height: 5;
}
```

The `split-pane-divider` substructure contains a grabber substructure, which is a `StackPane`. Its CSS style-class name is `horizontal-grabber` for a horizontal `SplitPane` and `vertical-grabber` for a vertical `SplitPane`. The grabber is shown in the middle of the divider.

## Understanding the *Slider* Control

A `Slider` lets the user select a numeric value from a numeric range graphically by sliding a thumb (or knob) along a track. A slider can be horizontal or vertical. Figure 12-58 shows a horizontal slider.



**Figure 12-58.** A horizontal *Slider* control and its parts

A slider has minimum and maximum values that determine the range of the valid selectable values. The thumb of the slider indicates its current value. You can slide the thumb along the track to change the current value. Major and minor tick marks show the location of values along the track. You can also show tick labels. Custom labels are also supported.

The following code creates a `Slider` control using its default constructor that sets 0, 100, and 0 as the minimum, maximum, and current value, respectively. The default orientation is horizontal.

```
// Create a horizontal slider
Slider s1 = new Slider();
```

Use another constructor to specify the minimum, maximum, and current values:

```
// Create a horizontal slider with the specified min, max, and
value
double min = 0.0;
double max = 200.0;
double value = 50.0;
Slider s2 = new Slider(min, max, value);
```

A `Slider` control contains several properties. I will discuss them by categories. The `s1` is horizontal.

orientation property specifies the orientation of the slider:

```
// Create a vertical slider
Slider vs = new Slider();
vs.setOrientation(Orientation.VERTICAL);
```

The following properties are related to the current value and the range of values:

- min
- max
- value
- valueChanging
- snapToTicks

The min, max, and value properties are double properties, and they represent the minimum, maximum, and current values, respectively, of the slider. The current value of the slider can be changed by dragging the thumb on the track or using the setValue() method. The following snippet of code creates a slider and sets its min, max, and value properties to 0, 10, and 3, respectively:

```
Slider scoreSlider = new Slider();
scoreSlider.setMin(0.0);
scoreSlider.setMax(10.0);
scoreSlider.setValue(3.0);
```

Typically, you want to perform an action when the value property of the slider changes. You will need to add a ChangeListener to the value property. The following statement adds a ChangeListener using a lambda expression to the scoreSlider control and prints the old and new values whenever the value property changes:

```
scoreSlider.valueProperty().addListener(
    (ObservableValue<? extends Number> prop, Number oldVal,
    Number newVal) -> {
        System.out.println("Changed from " + oldVal + " to "
+ newVal);
});
```

The valueChanging property is a boolean property. It is set to true when the user presses the thumb and is set to false when the thumb is released. As the user drags the thumb, the value keeps changing and the valueChanging property is true. This property helps you avoid repeating an action if you want to take the action only once when the value changes.

The snapToTicks property is a boolean property, which is false by default. It specifies whether the value property of the slider is always

aligned with the tick marks. If it is set to false, the value could be anywhere in the min to max range.

Be careful in using the valueChanging property inside a ChangeListener. The listener may be called several times for what the user sees as one change. Expecting that the ChangeListener will be notified when the valueChanging property changes from true to false, you wrap the main logic for the action inside an if statement:

```
if (scoreSlider.isValueChanging()) {
    // Do not perform any action as the value changes
} else {
    // Perform the action as the value has been changed
}
```

The logic works fine when the snapToTicks property is set to true. The ChangeListener for the value property is notified when the valueChanging property changes from true to false only when the snapToTicks property is set to true. Therefore, do not write the above logic unless you have set the snapToTicks property to true as well.

The following properties of the Slider class specify the tick spacing:

- majorTickUnit
- minorTickCount
- blockIncrement

The majorTickUnit property is a double property. It specifies the unit of distance between two major ticks. Suppose the min property is set to 0 and the majorTickUnit to 10. The slider will have major ticks at 0, 10, 20, 30, and so forth. An out-of-range value for this property disables the major ticks. The default value for the property is 25.

The minorTickCount property is an integer property. It specifies the number of minor ticks between two major ticks. The default value for the property is 3.

You can change the thumb position by using keys, for example, using left and right arrow keys in a horizontal slider and up and down arrow keys in a vertical slider. The blockIncrement property is a double property. It specifies the amount by which the current value of the slider is adjusted when the thumb is operating by using keys. The default value for the property is 10.

The following properties specify whether the tick marks and tick labels are shown; by default, they are set to false:

- showTickMarks
- showTickLabels

The `labelFormatter` property is an object property of the `StringConverter<Double>` type. By default, it is null and the slider uses a default `StringConverter` that displays the numeric values for the major ticks. The values for the major ticks are passed to the `toString()` method and the method is supposed to return a custom label for that value. The following snippet of code creates a slider with custom major tick labels, as shown in Figure 12-59:

```
Slider scoreSlider = new Slider();
scoreSlider.setShowTickLabels(true);
scoreSlider.setShowTickMarks(true);
scoreSlider.setMajorTickUnit(10);
scoreSlider.setMinorTickCount(3);
scoreSlider.setBlockIncrement(20);
scoreSlider.setSnapToTicks(true);

// Set a custom major tick formatter
scoreSlider.setLabelFormatter(new StringConverter<Double>() {
    @Override
    public String toString(Double value) {
        String label = "";
        if (value == 40) {
            label = "F";
        } else if (value == 70) {
            label = "C";
        } else if (value == 80) {
            label = "B";
        } else if (value == 90) {
            label = "A";
        }
        return label;
    }

    @Override
    public Double fromString(String string) {
        return null; // Not used
    }
});
```



**Figure 12-59.** A slider with custom major tick labels

The program in Listing 12-36 shows how to use `Slider` controls. It adds a `Rectangle`, a `Label`, and three `Slider` controls to a window. It adds a `ChangeListener` to the sliders. Sliders represent red, green, and blue components of a color. When you change the value for a slider, the new color is computed and set as the fill color for the rectangle.

### ***Listing 12-36.*** Using the Slider Control

```
// SliderTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.beans.value.ObservableValue;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.Slider;
import javafx.scene.layout.GridPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

public class SliderTest extends Application {
    Rectangle rect = new Rectangle(0, 0, 200, 50);
    Slider redSlider = getSlider();
    Slider greenSlider = getSlider();
    Slider blueSlider = getSlider();

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

    @Override
    public void start(Stage stage) {
        // Add a ChangeListener to all sliders
        redSlider.valueProperty().addListener(this::changed);
;
        greenSlider.valueProperty().addListener(this::change
d);
        blueSlider.valueProperty().addListener(this::changed
);
        GridPane root = new GridPane();
        root.setVgap(10);
        root.add(rect, 0, 0, 2, 1);
        root.add(new Label("Use sliders to change the fill
color"), 0, 1, 2, 1);
        root.addRow(2, new Label("Red:"), redSlider);
        root.addRow(3, new Label("Green:"), greenSlider);
        root.addRow(4, new Label("Blue:"), blueSlider);

        root.setStyle("-fx-padding: 10;" +
                     "-fx-border-style: solid inside;" +
                     "-fx-border-width: 2;" +
                     "-fx-border-insets: 5;" +
                     "-fx-border-radius: 5;" +
                     "-fx-border-color: blue;");
    }

    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("Using Slider Controls");
}
```

```
        stage.show();

        // Adjust the fill color of the rectangle
        changeColor();
    }

    public Slider getSlider() {
        Slider slider = new Slider(0, 255, 125);
        slider.setShowTickLabels(true);
        slider.setShowTickMarks(true);
        slider.setMajorTickUnit(85);
        slider.setMinorTickCount(10);
        slider.setBlockIncrement(20);
        slider.setSnapToTicks(true);
        return slider;
    }

    // A change listener to track the change in color
    public void changed(ObservableValue<? extends Number> prop,
                        Number oldValue,
                        Number newValue) {
        changeColor();
    }

    public void changeColor() {
        int r = (int)redSlider.getValue();
        int g = (int)greenSlider.getValue();
        int b = (int)blueSlider.getValue();
        Color fillColor = Color.rgb(r, g, b);
        rect.setFill(fillColor);
    }
}
```

## Styling *Slider* with CSS

The default CSS style-class name for a `Slider` control is `slider`. `Slider` contains the following CSS properties, each of them corresponds to its Java property in the `Slider` class:

- `-fx-orientation`
- `-fx-show-tick-labels`
- `-fx-show-tick-marks`
- `-fx-major-tick-unit`
- `-fx-minor-tick-count`
- `-fx-show-tick-labels`
- `-fx-snap-to-ticks`
- `-fx-block-increment`

Slider supports horizontal and vertical CSS pseudo-classes that apply to horizontal and vertical sliders, respectively. A Slider control contains three substructures that can be styled:

- axis
- track
- thumb

The axis substructure is a NumberAxis. It displays the tick marks and tick labels. The following code sets the tick label color to blue, major tick length to 15px, minor tick length to 5px, major tick color to red, and minor tick color to green:

```
.slider > .axis {
    -fx-tick-label-fill: blue;
    -fx-tick-length: 15px;
    -fx-minor-tick-length: 5px
}

.slider > .axis > .axis-tick-mark {
    -fx-stroke: red;
}

.slider > .axis > .axis-minor-tick-mark {
    -fx-stroke: green;
}
```

The track substructure is a StackPane. The following code changes the background color of track to red:

```
.slider > .track {
    -fx-background-color: red;
}
```

The thumb substructure is a StackPane. The thumb looks circular because it is given a background radius. If you remove the background radius, it will look rectangular, as shown in the following code:

```
.slider .thumb {
    -fx-background-radius: 0;
}
```

You can make an image like a thumb by setting the background of the thumb substructure to an image as follows (assuming that the thumb.jpg image file exists in the same directory as the CSS file containing the style):

```
.slider .thumb {
    -fx-background-image: url("thumb.jpg");
}
```

You can give the thumb any shape using the `-fx-shape` CSS property. The following code gives the thumb a triangular shape. The triangle is inverted for a horizontal slider and is pointed to the right for a vertical slider. Figure 12-60 shows a horizontal slider with the thumb.

```
/* An inverted triangle */
.slider > .thumb {
    -fx-shape: "M0, 0L10, 0L5, 10 z";
}

/* A triangle pointing to the right*/
.slider:vertical > .thumb {
    -fx-shape: "M0, 0L10, 5L0, 10 z";
}
```



**Figure 12-60.** A slider with an inverted triangle thumb

The following code gives the thumb a shape of a triangle placed beside a rectangle. The triangle is inverted for a horizontal slider and is pointed to the right for a vertical slider. Figure 12-61 shows a horizontal slider with the thumb.

```
/* An inverted triangle below a rectangle*/
.slider > .thumb {
    -fx-shape: "M0, 0L10, 0L10, 5L5, 10L0, 5 z";
}

/* A triangle pointing to the right by the right side of
a rectangle */
.slider:vertical > .thumb {
    -fx-shape: "M0, 0L5, 0L10, 5L5, 10L0, 10 z";
}
```



**Figure 12-61.** A slider with a thumb of an inverted triangle below a rectangle

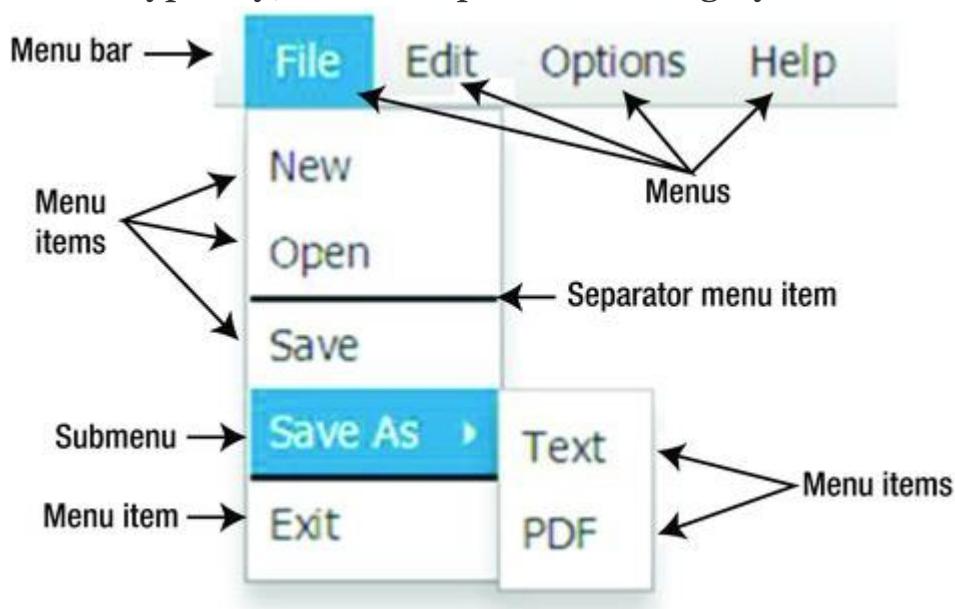
## Understanding Menus

A menu is used to provide a list of actionable items to the user in a compact form. You can also provide the same list of items using a group of buttons, where each button represents an actionable item. It is a matter of preference which one you use: a menu or a group of buttons. There is a noticeable advantage of using a menu. It uses much less space on the screen, compared to a group of buttons, by folding (or nesting) the group of items under another item. For example, if you have used a file editor, the menu items such as New, Open, Save, and Print are nested under a top-level File menu. A user needs to click the File menu to see the list of items that are available under it. Typically, in cases of a group of buttons, all items are visible to the user all the time, and it is

easy for users to know what actions are available. Therefore, there is little tradeoff between the amount of space and usability when you decide to use a menu or buttons. Typically, a menu bar is displayed at the top of a window.

**Tip** There is another kind of menu, which is called a *context menu* or *pop-up menu*, which is displayed on demand. I will discuss context menus in the next section.

A menu consists of several parts. Figure 12-62 shows a menu and its parts when the Save As submenu is expanded. A menu bar is the topmost part of the menu that holds menus. The menu bar is always visible. File, Edit, Options, and Help are the menu items shown in Figure 12-62. A menu contains menu items and submenus. In Figure 12-62, the File menu contains four menu items: New, Open, Save, and Exit; it contains two separator menu items and one Save As submenu. The Save As submenu contains two menu items: Text and PDF. A menu item is an actionable item. A separator menu item has a horizontal line that separates a group of related menu items from another group of items in a menu. Typically, a menu represents a category of items.



**Figure 12-62.** A menu with a menu bar, menus, submenus, separators, and menu items

Using a menu is a multistep process. The following sections describe the steps in detail. The following is the summary of steps:

1. Create a menu bar and add it to a container.
2. Create menus and add them to the menu bar.
3. Create menu items and add them to the menus.

4. Add `ActionEvent` handlers to the menu items to perform actions when they are clicked.

## Using Menu Bars

A menu bar is a horizontal bar that acts as a container for menus. An instance of the `MenuBar` class represents a menu bar. You can create a `MenuBar` using its default constructor:

```
MenuBar menuBar = new MenuBar();
```

`MenuBar` is a control. Typically, it is added to the top part of a window. If you use a `BorderPane` as the root for a scene in a window, the top region is the usual place for a `MenuBar`:

```
// Add the MenuBar to the top region
BorderPane root = new BorderPane();
root.setBottom(menuBar);
```

The `MenuBar` class contains a `useSystemMenuBar` property, which is of `boolean` type. By default, it is set to `false`. When set to `true`, it will use the system menu bar if the platform supports it. For example, Mac supports a system menu bar. If you set this property to `true` on Mac, the `MenuBar` will use the system menu bar to display its items:

```
// Let the MenuBar use system menu bar
menuBar.setUseSystemMenuBar(true);
```

A `MenuBar` itself does not take any space unless you add menus to it. Its size is computed based on the details of the menus it contains.

A `MenuBar` stores all of its menus in

an `ObservableList` of `Menu`s whose reference is returned by its `getMenus()` method:

```
// Add some menus to the MenuBar
Menu fileMenu = new Menu("File");
Menu editMenu = new Menu("Edit");
menuBar.getMenus().addAll(fileMenu, editMenu);
```

## Using Menus

A menu contains a list of actionable items, which are displayed on demand, for example, by clicking it. The list of menu items is hidden when the user selects an item or moves the mouse pointer outside the list. A menu is typically added to a menu bar or another menu as a submenu.

An instance of the `Menu` class represents a menu. A menu displays text and a graphic. Use the default constructor to create an empty menu, and later, set the text and graphic:

```
// Create a Menu with an empty string text and no graphic
Menu aMenu = new Menu();
```

```
// Set the text and graphic to the Menu  
aMenu.setText("Text");  
aMenu.setGraphic(new ImageView(new Image("image.jpg")));
```

You can create a menu with its text, or text and a graphic, using other constructors:

```
// Create a File Menu  
Menu fileMenu1 = new Menu("File");  
  
// Create a File Menu  
Menu fileMenu2 = new Menu("File", new ImageView(new  
Image("file.jpg")));
```

The `Menu` class is inherited from the `MenuItem` class, which is inherited from the `Object` class. `Menu` is not a node, and therefore, it cannot be added to a scene graph directly. You need to add it to a `MenuBar`. Use the `getMenus()` method to get the `ObservableList<Menu>` for the `MenuBar` and add instances of the `Menu` class to the list. The following snippet of code adds four `Menu` instances to a `MenuBar`:

```
Menu fileMenu = new Menu("File");  
Menu editMenu = new Menu("Edit");  
Menu optionsMenu = new Menu("Options");  
Menu helpMenu = new Menu("Help");  
  
// Add menus to a menu bar  
MenuBar menuBar = new MenuBar();  
menuBar.getMenus().addAll(fileMenu, editMenu, optionsMenu,  
helpMenu);
```

When a menu is clicked, typically its list of menu items are displayed, but no action is taken. The `Menu` class contains the following properties that can be set to handle when its list of options are showing, shown, hiding, and hidden, respectively:

- `onShowing`
- `onShown`
- `onHiding`
- `onHidden`
- `showing`

The `onShowing` event handler is called just before the menu items for the menu is shown. The `onShown` event handler is called after the menu items are displayed. The `onHiding` and `onHidden` event handlers are the counterparts of the `onShowing` and `onShown` event handlers, respectively.

Typically, you add an `onShowing` event handler that enables or disables its menu items based on some criteria. For example, suppose you have an `Edit` menu with `Cut`, `Copy`, and `Paste` menu items. In the `onShowing` event handler, you would enable or disable these menu

items depending on whether the focus is in a text input control, if the control is enabled, or if the control has selection:

```
editMenu.setOnAction(e -> {/* Enable/disable menu items here */});
```

**Tip** Users do not like surprises when using a GUI application. For a better user experience, you should disable menu items instead of making them invisible when they are not applicable. Making them invisible changes the positions of other items and users have to relocate them.

The `showing` property is a read-only boolean property. It is set to true when the items in the menu are showing. It is set to false when they are hidden.

The program in Listing 12-37 puts this all together. It creates four menus, a menu bar, adds menus to the menu bar, and adds the menu bar to the top region of a `BorderPane`. Figure 12-63 shows the menu bar in the window. But you have not seen anything exciting about menus yet! You will need to add menu items to the menus to experience some excitement.

### ***Listing 12-37.*** Creating a Menu Bar and Adding Menus to It

```
// MenuTest.java
package com.jdojo.control;

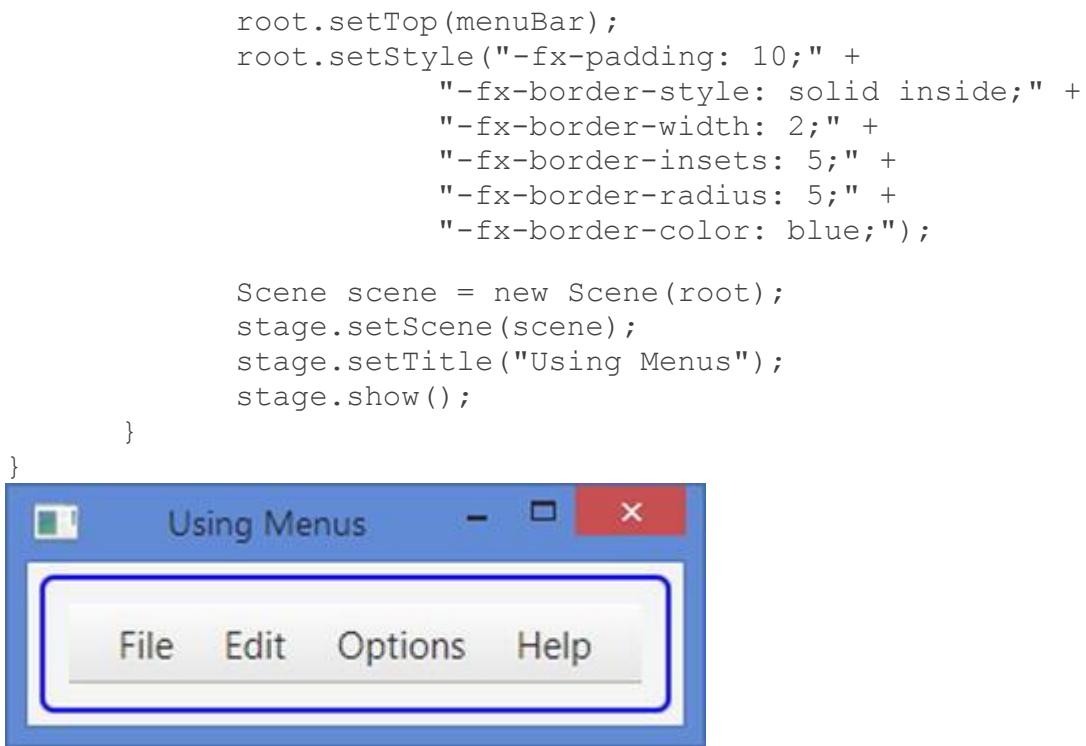
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Menu;
import javafx.scene.controlMenuBar;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class MenuTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // Create some menus
        Menu fileMenu = new Menu("File");
        Menu editMenu = new Menu("Edit");
        Menu optionsMenu = new Menu("Options");
        Menu helpMenu = new Menu("Help");

        // Add menus to a menu bar
        MenuBar menuBar = new MenuBar();
        menuBar.getMenus().addAll(fileMenu, editMenu,
        optionsMenu, helpMenu);

        BorderPane root = new BorderPane();
```



**Figure 12-63.** A menu bar with four menus

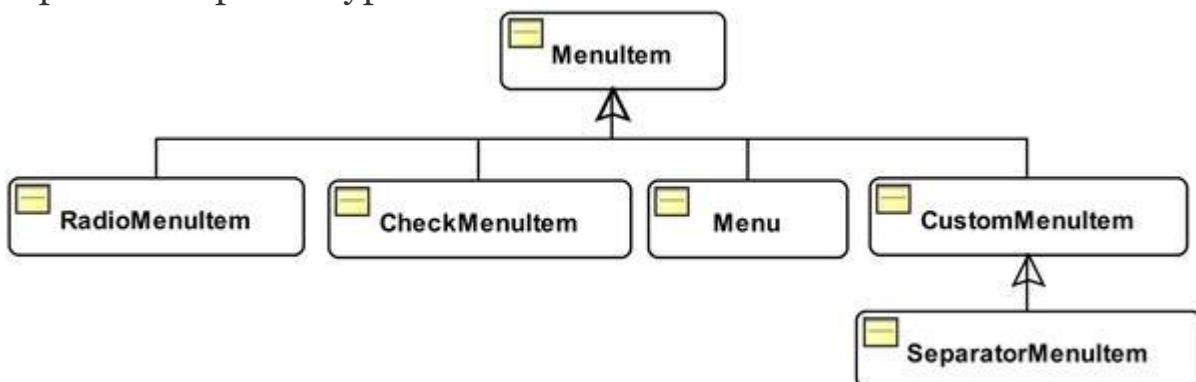
### Using Menu Items

A menu item is an actionable item in a menu. The action associated with a menu item is performed by the mouse or keys. Menu items can be styled using a CSS.

An instance of the `MenuItem` class represents a menu item.

The `MenuItem` class is not a node. It is inherited from the `Object` class and, therefore, cannot be added directly to a scene graph. You need to add it to a menu.

You can add several types of menu items to a menu. Figure 12-64 shows the class diagram for the `MenuItem` class and its subclasses that represent a specific type of menu item.



**Figure 12-64.** A class diagram for the `MenuItem` class and its subclasses

You can use the following types of menu items:

- A `MenuItem` for an actionable option
- A `RadioMenuItem` for a group of mutually exclusive options
- A `CheckMenuItem` for a toggle option
- A `Menu`, when used as a menu item and acts as a submenu that holds a list of menu items
- A `CustomMenuItem` for an arbitrary node to be used as an menu item
- A `SeparatorMenuItem`, which is a `CustomMenuItem`, to display a separator as a menu item

I will discuss all menu item types in details in the sections to follow.

## Using a `MenuItem`

A `MenuItem` represents an actionable option. When it is clicked, the registered `ActionEventHandlers` are called. The following snippet of code creates an `Exit MenuItem` and adds an `ActionEvent` handler that exits the application:

```
MenuItem exitItem = new MenuItem("Exit");
exitItem.setOnAction(e -> Platform.exit());
```

A `MenuItem` is added to a menu. A menu stores the reference of its `MenuItems` in an `ObservableList<MenuItem>` whose reference can be obtained using the `getItems()` method:

```
Menu fileMenu = new Menu("File");
fileMenu.getItems().add(exitItem);
```

The `MenuItem` class contains the following properties that apply to all types of menu items:

- `text`
- `graphic`
- `disable`
- `visible`
- `accelerator`
- `mnemonicParsing`
- `onAction`
- `onMenuValidation`
- `parentMenu`
- `parentPopup`
- `style`
- `id`

The `text` and `graphic` properties are the text and graphics for the menu item, respectively, which are of `String` and `Node` types.

The `disable` and `visible` properties are `boolean` properties. They specify whether the menu item is disabled and visible.

The `accelerator` property is an object property of the `KeyCombination` type that specifies a key combination that can be used to execute the action associated with the menu item in one keystroke. The following snippet of code creates a `Rectangle` menu item and sets its accelerator to `Alt + R`. The accelerator for a menu item is shown next to it, as shown in Figure 12-65, so the user can learn about it by looking at the menu item. The user can activate the `Rectangle` menu item directly by pressing `Alt + R`.

```
MenuItem rectItem = new MenuItem("Rectangle");
KeyCombination kr = new KeyCodeCombination(KeyCode.R,
KeyCombination.ALT_DOWN);
rectItem.setAccelerator(kr);
```

**Rectangle Alt+R**

**Figure 12-65.** A menu item with an accelerator `Alt + R`

The `mnemonicParsing` property is a `boolean` property. It enables or disables text parsing to detect a mnemonic character. By default, it is set to true for menu items. If it is set to true, the text for the menu item is parsed for an underscore character. The character following the first underscore is added as the mnemonic for the menu item. Pressing the `Alt` key on Windows highlights mnemonics for all menu items. Typically, mnemonic characters are shown in underlined font style. Pressing the key for the mnemonic character activates the menu item.

```
// Create a menu item with x as its mnemonic character
MenuItem exitItem = new MenuItem("E_xit");
```

The `onAction` property is an `ActionEvent` handler that is called when the menu item is activated, for example, by clicking it with a mouse or pressing its accelerator key:

```
// Close the application when the Exit menu item is activated
exitItem.setOnAction(e -> Platform.exit());
```

The `onMenuValidation` property is an event handler that is called when a `MenuItem` is accessed using its accelerator or when the `onShowing` event handler for its menu (the parent) is called. For a menu, this handler is called when its menu items are shown.

The `parentMenu` property is a read-only object property of the `MenuItem` type. It is the reference of the `Menu`, which contains the menu item. Using this property and the `items` list returned by

the `getItems()` method of the `Menu` class, you can navigate the menu tree from top to bottom and vice versa.

The `parentPopup` property is a read-only object property of the `ContextMenu` type. It is the reference of the `ContextMenu` in which the menu item appears. It is `null` for a menu item appearing in a normal menu.

The style and ID properties are included to support styling using a CSS. They represent the CSS style and ID.

## Using a `RadioMenuItem`

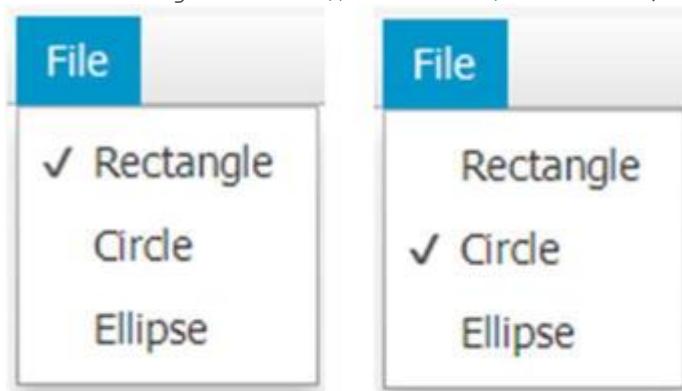
A `RadioMenuItem` represents a mutually exclusive option. Typically, you add `RadioMenuItem` in multiples to a `ToggleGroup`, so only one item is selected. `RadioMenuItem` displays a check mark when selected. The following snippet of code creates three instances of `RadioMenuItem` and adds them to a `ToggleGroup`. Finally, they are all added to a File Menu. Typically, a `RadioMenuItem` in a group is selected by default. Figure 12-66 shows the group of `RadioMenuItem`s: once when Rectangle is selected and once when Circle is selected.

```
// Create three RadioMenuItem
RadioMenuItem rectItem = new RadioMenuItem("Rectangle");
RadioMenuItem circleItem = new RadioMenuItem("Circle");
RadioMenuItem ellipseItem = new RadioMenuItem("Ellipse");

// Select the Rantangle option by default
rectItem.setSelected(true);

// Add them to a ToggleGroup to make them mutually exclusive
ToggleGroup shapeGroup = new ToggleGroup();
shapeGroup.getToggles().addAll(rectItem, circleItem,
ellipseItem);

// Add RadioMenuItem to a File Menu
Menu fileMenu = new Menu("File");
fileMenu.getItems().addAll(rectItem, circleItem, ellipseItem);
```



**Figure 12-66.** `RadioMenuItem` in action

Add an `ActionEvent` handler to the `RadioMenuItem` if you want to perform an action when it is selected. The following snippet of code adds an `ActionEvent` handler to each `RadioMenuItem`, which calls a `draw()` method:

```
rectItem.setOnAction(e -> draw());
circleItem.setOnAction(e -> draw());
ellipseItem.setOnAction(e -> draw());
```

## Using a `CheckMenuItem`

Use a `CheckMenuItem` to represent a boolean menu item that can be toggled between selected and unselected states. Suppose you have an application that draws shapes. You can have a Draw Stroke menu item as a `CheckMenuItem`. When it is selected, a stroke will be drawn for the shape. Otherwise, the shape will not have a stroke, as indicated in the following code. Use an `ActionEvent` handler to be notified when the state of the `CheckMenuItem` is toggled.

```
CheckMenuItem strokeItem = new CheckMenuItem("Draw Stroke");
strokeItem.setOnAction( e -> drawStroke());
```

When a `CheckMenuItem` is selected, a check mark is displayed beside it.

## Using a Submenu Item

Notice that the `Menu` class is inherited from the `MenuItem` class. This makes it possible to use a `Menu` in place of a `MenuItem`. Use a `Menu` as a menu item to create a submenu. When the mouse hovers over a submenu, its list of options is displayed.

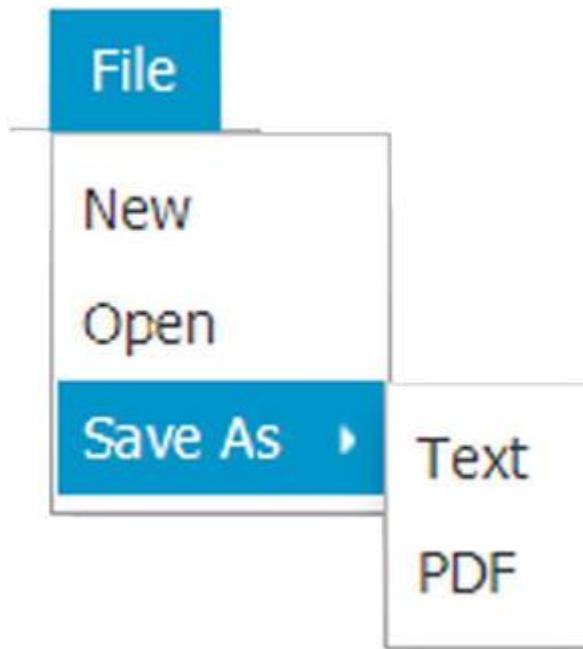
The following snippet of code creates a `MenuBar`, adds a File menu, adds New and Open `MenuItem`s and a Save As submenu to the File menu, and adds Text and PDF menu items to the Save As submenu. It produces a menu as shown in Figure 12-67.

```
MenuBar menuBar = new MenuBar();
Menu fileMenu = new Menu("File");
menuBar.getMenus().addAll(fileMenu);

MenuItem newItem = new MenuItem("New");
MenuItem openItem = new MenuItem("Open");
Menu saveAsSubMenu = new Menu("Save As");

// Add menu items to the File menu
fileMenu.getItems().addAll(newItem, openItem, saveAsSubMenu);

MenuItem textItem = new MenuItem("Text");
MenuItem pdfItem = new MenuItem("PDF");
saveAsSubMenu.getItems().addAll(textItem, pdfItem);
```



**Figure 12-67.** A menu used as a submenu

Typically, you do not add an `ActionEvent` handler for a submenu. Rather, you set an event handler to the `onShowing` property that is called before the list of items for the submenu is displayed. The event handler is used to enable or disable menu items.

### Using a `CustomMenuItem`

`CustomMenuItem` is a simple yet powerful menu item type. It opens the door for all kinds of creativity for designing menu items. It lets you use any node. For example, you can use a `Slider`, a `TextField`, or an `HBox` as a menu item. The `CustomMenuItem` class contains two properties:

- `content`
- `hideOnClick`

The `content` property is an object property of `Node` type. Its value is the node that you want to use as the menu item.

When you click a menu item, all visible menus are hidden and only top-level menus in the menu bar stay visible. When you use a custom menu item that has controls, you do not want to hide menus when the user clicks it because the user needs to interact with the menu item, for example, to enter or select some data. The `hideOnClick` property is a `boolean` property that lets you control this behavior. By default, it is

set to true, which means clicking a custom menu hides all showing menus.

The `CustomMenuItem` class provides several constructors. The default constructor creates a custom menu item setting the `content` property to null and the `hideOnClick` property to true, as shown in the following code:

```
// Create a Slider control
Slider slider = new Slider(1, 10, 1);

// Create a custom menu item and set its content and hideOnClick
properties
CustomMenuItem cmi1 = new CustomMenuItem();
cmi1.setContent(slider);
cmi1.setHideOnClick(false);

// Create a custom menu item with a Slider content and
// set the hideOnClick property to false
CustomMenuItem cmi2 = new CustomMenuItem(slider);
cmi1.setHideOnClick(false);

// Create a custom menu item with a Slider content and false
hideOnClick
CustomMenuItem cmi2 = new CustomMenuItem(slider, false);
```

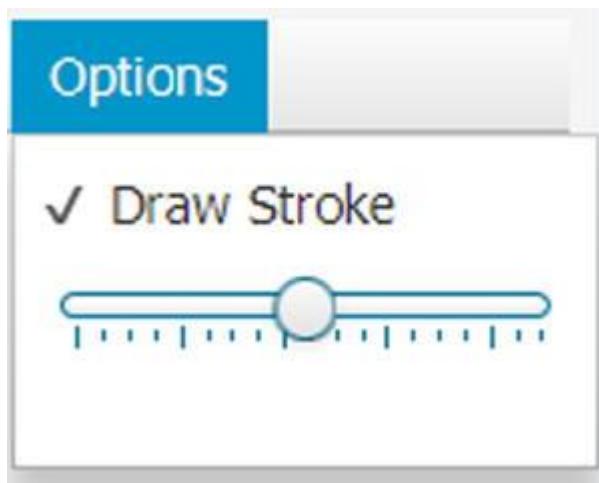
The following snippet of code produces a menu as shown in Figure 12-68. One of the menu items is a `CustomMenuItem`, which uses a `slider` as its content.

```
CheckMenuItem strokeItem = new CheckMenuItem("Draw Stroke");
strokeItem.setSelected(true);

Slider strokeWidthSlider = new Slider(1, 10, 1);
strokeWidthSlider.setShowTickLabels(true);
strokeWidthSlider.setShowTickMarks(true);
strokeWidthSlider.setMajorTickUnit(2);
CustomMenuItem strokeWidthItem = new
CustomMenuItem(strokeWidthSlider, false);

Menu optionsMenu = new Menu("Options");
optionsMenu.getItems().addAll(strokeItem, strokeWidthItem);

MenuBar menuBar = newMenuBar();
menuBar.getMenus().add(optionsMenu);
```



**Figure 12-68.** A slider as a custom menu item

### Using a **SeparatorMenuItem**

There is nothing special to discuss about the **SeparatorMenuItem**. It inherits from the **CustomMenuItem**. It uses a horizontal **Separator** control as its content and sets the `hideOnClick` to false. It is used to separate menu items belonging to different groups, as shown in the following code. It provides a default constructor.

```
// Create a separator menu item
SeparatorMenuItem smi = SeparatorMenuItem();
```

### Putting All Parts of Menus Together

Understanding the parts of menus is easy. However, using them in code is tricky because you have to create all parts separately, add listeners to them, and then assemble them.

The program in Listing 12-38 creates a shape drawing application using menus. It uses all types of menu items. The program displays a window with a **BorderPane** as the root of its scene. The top region contains a menu and the center region contains a canvas on which shapes are drawn.

Run the application and use the File menu to draw different types of shapes; clicking the Clear menu item clears the canvas. Clicking the Exit menu item closes the application.

Use the Options menu to draw or not to draw the strokes and set the stroke width. Notice that a slider is used as a custom menu item under the Options menu. When you adjust the slider value, the stroke width of the drawn shape is adjusted accordingly. The Draw Stroke menu item is a **CheckMenuItem**. When it is unselected, the slider menu item is disabled and the shape does not use a stroke.

**Listing 12-38.** Using Menus in a Shape Drawing Application

```
// MenuItemTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.application.Platform;
import javafx.beans.value.ObservableValue;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.control.CheckMenuItem;
import javafx.scene.control.CustomMenuItem;
import javafx.scene.control.Menu;
import javafx.scene.controlMenuBar;
import javafx.scene.control.MenuItem;
import javafx.scene.control.RadioButton;
import javafx.scene.control.SeparatorMenuItem;
import javafx.scene.control.Slider;
import javafx.scene.control.ToggleGroup;
import javafx.scene.input.KeyCode;
import javafx.scene.input.KeyCodeCombination;
import javafx.scene.input.KeyCombination;
import javafx.scene.layout.BorderPane;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class MenuItemTest extends Application {
    // A canvas to draw shapes
    Canvas canvas = new Canvas(200, 200);

    // Create three RadioMenuItem for shapes
    RadioMenuItem rectItem = new RadioMenuItem("_Rectangle");
    RadioMenuItem circleItem = new RadioMenuItem("_Circle");
    RadioMenuItem ellipseItem = new RadioMenuItem("_Ellipse");

    // A menu item to draw stroke
    CheckMenuItem strokeItem = new CheckMenuItem("Draw
_Stroke");

    // To adjust the stroke width
    Slider strokeWidthSlider = new Slider(1, 10, 1);
    CustomMenuItem strokeWidthItem = new
CustomMenuItem(strokeWidthSlider, false);

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

    @Override
    public void start(Stage stage) {
```

```

Menu fileMenu = getFileMenu();
Menu optionsMenu = getOptionsMenu();

MenuBar menuBar = new MenuBar();
menuBar.getMenus().addAll(fileMenu, optionsMenu);

// Draw the default shape, which is a Rectangle
this.draw();

BorderPane root = new BorderPane();
root.setTop(menuBar);
root.setCenter(canvas);
root.setStyle("-fx-padding: 10;" +
    "-fx-border-style: solid inside;" +
    "-fx-border-width: 2;" +
    "-fx-border-insets: 5;" +
    "-fx-border-radius: 5;" +
    "-fx-border-color: blue;");

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("Using Different Types of Menu
Items");
stage.show();
}

public void draw() {
    GraphicsContext gc = canvas.getGraphicsContext2D();
    gc.clearRect(0, 0, 200, 200); // First clear the
canvas

    // Set drawing parameters
    gc.setFill(Color.TAN);
    gc.setStroke(Color.RED);
    gc.setLineWidth(strokeWidthSlider.getValue());

    String shapeType = getSelectedShape();
    switch(shapeType) {
        case "Rectangle":
            gc.fillRect(0, 0, 200, 200);
            if (strokeItem.isSelected()) {
                gc.strokeRect(0, 0, 200, 200);
            }
            break;
        case "Circle":
            gc.fillOval(10, 10, 180, 180);
            if (strokeItem.isSelected()) {
                gc.strokeOval(10, 10, 180, 180);
            }
            break;
        case "Ellipse":
            gc.fillOval(10, 10, 180, 150);
            if (strokeItem.isSelected()) {
                gc.strokeOval(10, 10, 180, 150);
            }
    }
}

```

```

        break;
    default:
        clear(); // Do not know the shape type
    }
}

public void clear() {
    canvas.getGraphicsContext2D().clearRect(0, 0, 200,
200);
    this.rectItem.setSelected(false);
    this.circleItem.setSelected(false);
    this.ellipseItem.setSelected(false);
}

public Menu getFileMenu() {
    Menu fileMenu = new Menu("_File");

    // Make Rectangle the default option
    rectItem.setSelected(true);

    // Set Key Combinations for shapes
    KeyCombination kr =
        new KeyCodeCombination(KeyCode.R,
KeyCombination.ALT_DOWN);
    KeyCombination kc =
        new KeyCodeCombination(KeyCode.C,
KeyCombination.ALT_DOWN);
    KeyCombination ke =
        new KeyCodeCombination(KeyCode.E,
KeyCombination.ALT_DOWN);
    rectItem.setAccelerator(kr);
    circleItem.setAccelerator(kc);
    ellipseItem.setAccelerator(ke);

    // Add ActionEvent handler to all shape radio menu
    items
    rectItem.setOnAction(e -> draw());
    circleItem.setOnAction(e -> draw());
    ellipseItem.setOnAction(e -> draw());

    // Add RadioMenuItem s to a ToggleGroup to make them
    mutually exclusive
    ToggleGroup shapeGroup = new ToggleGroup();
    shapeGroup.getToggles().addAll(rectItem, circleItem,
ellipseItem);

    MenuItem clearItem = new MenuItem("Cle_ar");
    clearItem.setOnAction(e -> clear());

    MenuItem exitItem = new MenuItem("E_xit");
    exitItem.setOnAction(e -> Platform.exit());

    // Add menu items to the File menu
    fileMenu.getItems().addAll(rectItem,
circleItem, ellipseItem,

```

```

        new SeparatorMenuItem(),
        clearItem,
        new SeparatorMenuItem(),
        exitItem);

    return fileMenu;
}

public Menu getOptionsMenu() {
    // Draw stroke by default
    strokeItem.setSelected(true);

    // Redraw the shape when draw stroke option toggles
    strokeItem.setOnAction(e -> syncStroke());

    // Configure the slider
    strokeWidthSlider.setShowTickLabels(true);
    strokeWidthSlider.setShowTickMarks(true);
    strokeWidthSlider.setMajorTickUnit(2);
    strokeWidthSlider.setSnapToPixel(true);
    strokeWidthSlider.valueProperty().addListener(this::strokeWidthChanged);

    Menu optionsMenu = new Menu("_Options");
    optionsMenu.getItems().addAll(strokeItem,
        this.strokeWidthItem);

    return optionsMenu;
}

public void strokeWidthChanged (ObservableValue<? extends Number> prop,
                               Number oldValue,
                               Number newValue) {
    draw();
}

public String getSelectedShape() {
    if (rectItem.isSelected()) {
        return "Rectangle";
    }
    else if (circleItem.isSelected()) {
        return "Circle";
    }
    else if (ellipseItem.isSelected()) {
        return "Ellipse";
    } else {
        return "";
    }
}

public void syncStroke() {
    // Enable/disable the slider
    strokeWidthSlider.setDisable(!strokeItem.isSelected());
}

draw();

```

```

    }
}

```

## Styling Menus Using CSS

There are several components involved in using a menu. Table 12-6 lists the default CSS style-class names for components related to menus.

**Table 12-6. CSS Default Style-Class Names for Menu-Related Components**

Menu Component	Style-Class Name
MenuBar	menu-bar
Menu	menu
MenuItem	menu-item
RadioMenuItem	radio-menu-item
CheckMenuItem	check-menu-item
CustomMenuItem	custom-menu-item
SeparatorMenuItem	separator-menu-item

MenuBar supports an `-fx-use-system-menu-bar` property, which is set to false by default. It indicates whether to use a system menu for the menu bar. It contains a menu substructure that holds the menus for the menu bar. Menu supports a showing CSS pseudo-class, which applies when the menu is showing. RadioMenuItem and CheckMenuItem support a selected CSS pseudo-class, which applies when the menu items are selected.

You can style several components of menus. Please refer to the modena.css file for the sample styles.

## Understanding the **ContextMenu** Control

ContextMenu is a pop-up control that displays a list of menu items on request. It is also known as a *context* or *pop-up* menu. By default, it is hidden. The user has to make a request, usually by right-clicking the

mouse button, to show it. It is hidden once a selection is made. The user can dismiss a context menu by pressing the Esc key or clicking outside its bounds.

A context menu has a usability problem. It is difficult for users to know about its existence. Usually, nontechnical users are not accustomed to right-clicking the mouse and making selections. For those users, you can present the same options using toolbars or buttons instead. Sometimes, a text message is included on the screen stating that the user needs to right-click to view or show the context menu.

An object of the `ContextMenu` class represents a context menu. It stores the reference of its menu items in an `ObservableList<MenuItem>`. The `getItems()` method returns the reference of the observable list. You will use the following three menu items in the examples presented below. Note that the menu items in a context menu could be an object of the `MenuItem` class or its subclasses. For the complete list of menu item types, please refer to the “Understanding Menus” section.

```
MenuItem rectItem = new MenuItem("Rectangle");
MenuItem circleItem = new MenuItem("Circle");
MenuItem ellipseItem = new MenuItem("Ellipse");
```

The default constructor of the `ContextMenu` class creates an empty menu. You need to add the menu items later:

```
ContextMenu ctxMenu = new ContextMenu();
ctxMenu.getItems().addAll(rectItem, circleItem, ellipseItem);
```

You can use the other constructor to create a context menu with an initial list of menu items:

```
ContextMenu ctxMenu = new ContextMenu(rectItem, circleItem,
ellipseItem);
```

Typically, context menus are provided for controls for accessing their commonly used features, for example, Cut, Copy, and Paste features of text input controls. Some controls have default context menus. The control class makes it easy to display a context menu. It has a `contextMenu` property. You need to set this property to your context menu reference for the control. The following snippet of code sets the context menu for a `TextField` control:

```
ContextMenu ctxMenu = ...
TextField nameFld = new TextField();
nameFld.setContextMenu(ctxMenu);
```

When you right-click the `TextField`, your context menu will be displayed instead the default one.

**Tip** Activating an empty context menu does not show anything. If you want to disable the default context menu for a control, set its `contextMenu` property to an empty `ContextMenu`.

Nodes that are not controls do not have a `contextMenu` property. You need to use the `show()` method of the `ContextMenu` class to display the context menu for these nodes. The `show()` method gives you full control of the position where the context menu is displayed. You can use it for controls as well if you want to finetune the positioning of the context menu. The `show()` method is overloaded:

```
void show(Node anchor, double screenX, double screenY)  
void show(Node anchor, Side side, double dx, double dy)
```

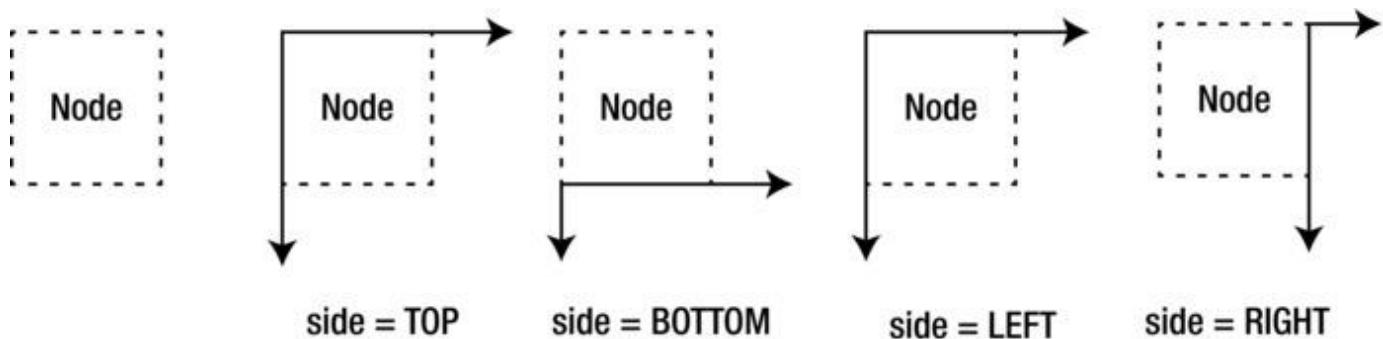
The first version takes the node for which the context menu is to be displayed with the x and y coordinates relative to the screen. Typically, you display a context menu in the mouse-clicked event where the `MouseEvent` object provides you the coordinates of the mouse pointer relative to the screen through the `getScreenX()` and `getScreenY()` methods.

The following snippet of code shows a context menu for a canvas at (100, 100) relative to the screen coordinate system:

```
Canvas canvas = ...  
ctxMenu.show(canvas, 100, 100);
```

The second version lets you finetune the position of the context menu relative to the specified `anchor` node. The `side` parameter specifies on which side of the `anchor` node the context menu is displayed. The possible values are one of the constants—`TOP`, `RIGHT`, `BOTTOM`, and `LEFT`—of the `Side` enum. The `dx` and `dy` parameters specify the x and y coordinates, respectively, relative to the `anchor` node coordinate system. This version of the `show()` method requires a little more explanation.

The `side` parameter has an effect of shifting the x axis and y axis of the `anchor` node. The `dx` and `dy` parameters are applied after the axes are shifted. Note that the axes are shifted only for computing the position of the context menu when this version of the method is called. They are not shifted permanently, and the `anchor` node position does not change at all. Figure 12-69 shows an anchor node and its x and y axes for the values of the `side` parameter. The `dx` and `dy` parameters are the x and y coordinates of the point relative to the shifted x axis and y axis of the node.



**Figure 12-69.** Shifting the x axis and y axis of the anchor node with the side parameter value

Note that the `LEFT` and `RIGHT` values for the `side` parameter are interpreted based on the node orientation of the anchor node. For a node orientation of `RIGHT_TO_LEFT`, the `LEFT` value means the right side of the node.

When you specify `TOP`, `LEFT`, or `null` for the `side` parameter, the `dx` and `dy` parameters are measured relative to the original x and y axes of the node. When you specify `BOTTOM` for the `side` parameter, the bottom of the node becomes the new x axis and the y axis remains the same. When you specify `RIGHT` for the `side` parameter, the right side of the node becomes the new y axis and the x axis remains the same.

The following call to the `show()` method displays a context menu at the upper left corner of the anchor node. The value of `Side.LEFT` or `null` for the `side` parameter would display the context menu at the same location:

```
ctxMenu.show(anchor, Side.TOP, 0, 0);
```

The following call to the `show()` method displays a context menu at the lower left corner of the anchor node:

```
ctxMenu.show(anchor, Side.BOTTOM, 0, 0);
```

Values for `dx` and `dy` can be negative. The following call to the `show()` method displays a context menu 10px above the upper left corner of the anchor node:

```
ctxMenu.show(myAnchor, Side.TOP, 0, -10);
```

The `hide()` method of the `ContextMenu` class hides the context menu, if it was showing. Typically, the context menu is hidden when you select a menu item. You need to use the `hide()` method when the context menu uses a custom menu item with `hideOnClick` property set to true. Typically, an `ActionEvent` handler is added to the menu items of a context menu. The `ContextMenu` class contains an `onAction` property, which is an `ActionEvent` handler. The `ActionEvent` handler, if set, for a `ContextMenu` is called every time a menu item is activated. You

can use this `ActionEvent` to execute a follow-up action when a menu item is activated.

The program in Listing 12-39 shows how to use a context menu. It displays a Label and a Canvas. When you right-click the canvas, a context menu with three menu items—Rectangle, Circle, and Ellipse—is displayed. Selecting one of the shapes from the menu items draws the shape on the canvas. The context menu is displayed when the mouse pointer is clicked.

**Listing 12-39.** Using the ContextMenu Control

```

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using Context Menus");
        stage.show();
    }

    public void showContextMenu(MouseEvent me) {
        // Show menu only on right click
        if (me.getButton() == MouseButton.SECONDARY) {
            MenuItem rectItem = new MenuItem("Rectangle");
            MenuItem circleItem = new MenuItem("Circle");
            MenuItem ellipseItem = new
MenuItem("Ellipse");
            rectItem.setOnAction(e -> draw("Rectangle"));
            circleItem.setOnAction(e -> draw("Circle"));
            ellipseItem.setOnAction(e -> draw("Ellipse"));
            ContextMenu ctxMenu =
                new ContextMenu(rectItem,
            circleItem, ellipseItem);
            ctxMenu.show(canvas, me.getScreenX(),
me.getScreenY());
        }
    }

    public void draw(String shapeType) {
        GraphicsContext gc = canvas.getGraphicsContext2D();
        gc.clearRect(0, 0, 200, 200); // clear the canvas
first
        gc.setFill(Color.TAN);

        if (shapeType.equals("Rectangle")) {
            gc.fillRect(0, 0, 200, 200);
        } else if (shapeType.equals("Circle")) {
            gc.fillOval(0, 0, 200, 200);
        } else if (shapeType.equals("Ellipse")) {
            gc.fillOval(10, 40, 180, 120);
        }
    }
}
}

```

## Styling *ContextMenu* with CSS

The default CSS style-class name for a *ContextMenu* is `context-menu`. Please refer to the `modena.css` file for sample styles for customizing the appearance of context menus. By default, a context menu uses a drop shadow effect. The following style sets the font size to 8pt and removes the default effect:

```

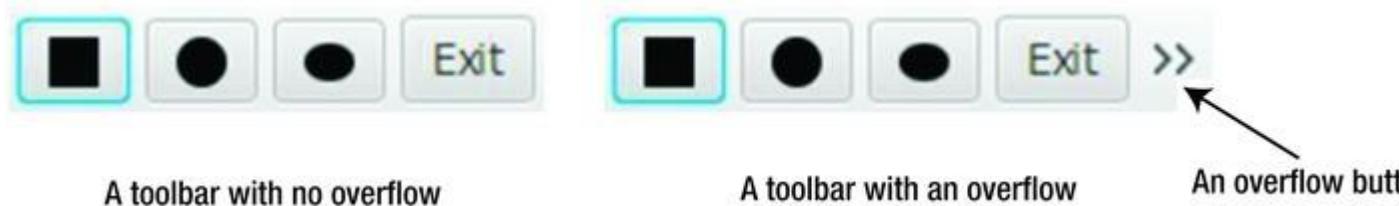
.context-menu {
    -fx-font-size: 8pt;
    -fx-effect: null;
}

```

## Understanding the **ToolBar** Control

ToolBar is used to display a group of nodes, which provide the commonly used action items on a screen. Typically, a ToolBar control contains the commonly used items that are also available through a menu and a context menu.

A ToolBar control can hold many types of nodes. The most commonly used nodes in a ToolBar are buttons and toggle buttons. Separators are used to separate a group of buttons from others. Typically, buttons are kept smaller by using small icons, preferably 16px by 16px in size. If the items in a toolbar overflow, an overflow button appears to allow users to navigate to the hidden items. A toolbar can have the orientation of horizontal or vertical. A horizontal toolbar arranges the items horizontally in one row. A vertical toolbar arranges the items in one column. Figure 12-70 shows two toolbars: one has no overflow and one has an overflow. The one with an overflow displays an overflow button (">>>). When you click the overflow button, the hidden toolbar items are displayed for selection.



**Figure 12-70.** A horizontal toolbar with three buttons

You will use the following four ToolBar items in the examples in this chapter:

```
Button rectBtn = new Button("", new Rectangle(0, 0, 16, 16));
Button circleBtn = new Button("", new Circle(0, 0, 8));
Button ellipseBtn = new Button("", new Ellipse(8, 8, 8, 6));
Button exitBtn = new Button("Exit");
```

A ToolBar control stores the reference of items in an ObservableList<Node>. Use the `getItems()` method to get the reference of the observable list.

The default constructor of the ToolBar class creates an empty toolbar:

```
ToolBar toolBar = new ToolBar();
toolBar.getItems().addAll(circleBtn, ellipseBtn, new Separator(),
exitBtn);
```

The ToolBar class provides another constructor that lets you add items:

```
ToolBar toolBar = new ToolBar(rectBtn, circleBtn, ellipseBtn,
new Separator(),
exitBtn);
```

The `orientation` property of the `ToolBar` class specifies its orientation: horizontal or vertical. By default, a toolbar uses the horizontal orientation. The following code sets it to vertical:

```
// Create a ToolBar and set its orientation to VERTICAL
ToolBar toolBar = new ToolBar();
toolBar.setOrientation(Orientation.VERTICAL);
```

**Tip** The orientation of a separator in a toolbar is automatically adjusted by the default CSS. It is good practice to provide tool tips for items in a toolbar, as they are small in size and typically do not use text content.

The program in Listing 12-40 shows how to create and use `ToolBar` controls. It creates a toolbar and adds four items. When you click one of the items with a shape, it draws the shape on a canvas. The Exit item closes the application.

### ***Listing 12-40.*** Using the `ToolBar` Control

```
// ToolBarTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.Separator;
import javafx.scene.control.ToolBar;
import javafx.scene.control.Tooltip;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Ellipse;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

public class ToolBarTest extends Application {
    // A canvas to draw shapes
    Canvas canvas = new Canvas(200, 200);

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

    public void start(Stage stage) {
        // Create ToolBar items
        Button rectBtn = new Button("", new Rectangle(0, 0,
16, 16));
```

```

        Button circleBtn = new Button("", new Circle(0, 0,
8));
        Button ellipseBtn = new Button("", new Ellipse(8, 8,
8, 6));
        Button exitBtn = new Button("Exit");

        // Set tooltips
        rectBtn.setTooltip(new Tooltip("Draws
a rectangle"));
        circleBtn.setTooltip(new Tooltip("Draws a circle"));
        ellipseBtn.setTooltip(new Tooltip("Draws an
ellipse"));
        exitBtn.setTooltip(new Tooltip("Exits
application"));

        // Add ActionEvent handlers for items
        rectBtn.setOnAction(e -> draw("Rectangle"));
        circleBtn.setOnAction(e -> draw("Circle"));
        ellipseBtn.setOnAction(e -> draw("Ellipse"));
        exitBtn.setOnAction(e -> Platform.exit());

        ToolBar toolBar = new ToolBar(rectBtn, circleBtn,
ellipseBtn,
                                         new Separator(),
                                         exitBtn);
        BorderPane root = new BorderPane();
        root.setTop(new VBox(new Label("Click a shape to
draw."), toolBar));
        root.setCenter(canvas);
        root.setStyle("-fx-padding: 10;" +
"-fx-border-style: solid inside;" +
"-fx-border-width: 2;" +
"-fx-border-insets: 5;" +
"-fx-border-radius: 5;" +
"-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using ToolBar Controls");
        stage.show();
    }

    public void draw(String shapeType) {
        GraphicsContext gc = canvas.getGraphicsContext2D();
        gc.clearRect(0, 0, 200, 200); // First clear the
canvas
        gc.setFill(Color.TAN);

        if (shapeType.equals("Rectangle")) {
            gc.fillRect(0, 0, 200, 200);
        } else if (shapeType.equals("Circle")) {
            gc.fillOval(0, 0, 200, 200);
        } else if (shapeType.equals("Ellipse")) {
            gc.fillOval(10, 40, 180, 120);
        }
    }
}

```

```
        }
    }
}
```

## Styling a Toolbar with CSS

The default CSS style-class name for a `ToolBar` is `tool-bar`. It contains an `-fx-orientation` CSS property that specifies its orientation with the possible values of *horizontal* and *vertical*. It supports horizontal and vertical CSS pseudo-classes that apply when its orientation is horizontal and vertical, respectively.

A toolbar uses a container to arrange the items. The container is an `HBox` for a horizontal orientation and a `VBox` for a vertical orientation. The CSS style-class name for the container is `container`. You can use all CSS properties for the `HBox` and `VBox` for the container. The `-fx-spacing` CSS property specifies the spacing between two adjacent items in the container. You can set this property for the toolbar or the container. Both of the following styles have the same effect on a horizontal toolbar:

```
.tool-bar {
    -fx-spacing: 2;
}

.tool-bar > .container {
    -fx-spacing: 2;
}
```

A toolbar contains a `tool-bar-overflow-button` substructure to represent the overflow button. It is a `StackPane`. The `tool-bar-overflow-button` contains an `arrow` substructure to represent the arrow in the overflow button. It is also a `StackPane`.

## Understanding `TabPane` and `Tab`

A window may not have enough space to display all of the pieces of information in one page view. JavaFX provides several controls to break down large content into multiple pages, for example, `Accordion` and `Pagination` controls. `TabPane` and `Tab` let you present information in a page much better. A `Tab` represents a page and a `TabPane` contains the `Tab`.

A `Tab` is not a control. An instance of the `Tab` class represents a `Tab`. The `Tab` class inherits from the `Object` class. However, the `Tab` supports some features as controls do, for example, they can be disabled, styled using CSS, and can have context menus and tool tips.

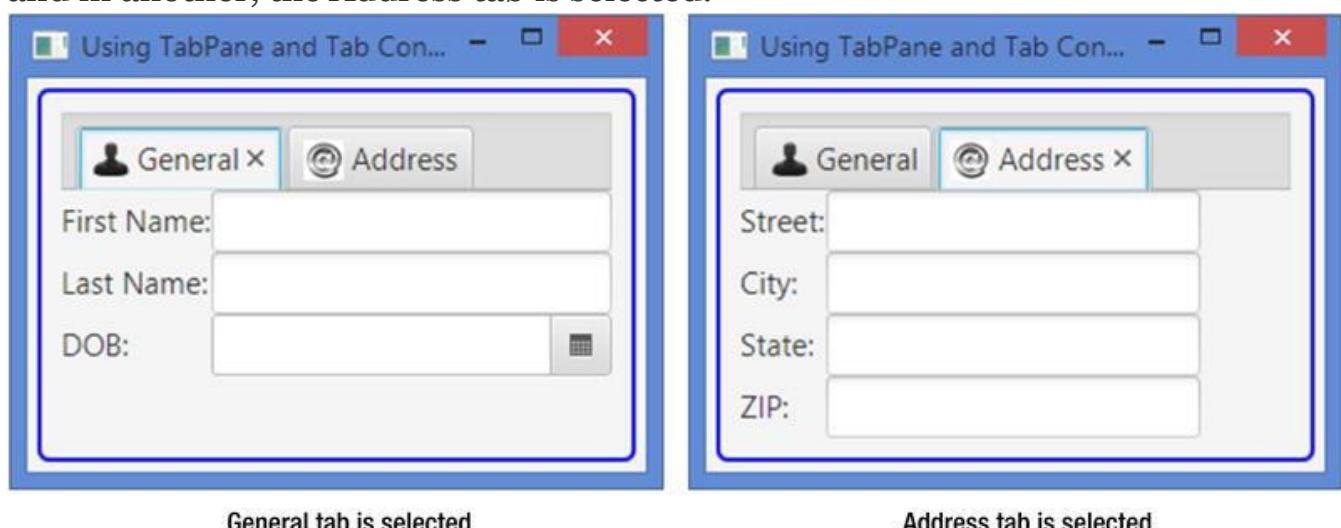
A `Tab` consists of a title and content. The title consists of text, an optional graphic, and an optional close button to close the tab. The

content consists of controls. Typically, controls are added to a layout pane, which is added to the Tab as its content.

Typically, the titles of the Tab in a TabPane are visible. The content area is shared by all Tabs. You need to select a Tab, by clicking its title, to view its content. You can select only one tab at a time in a TabPane. If the titles of all tabs are not visible, a control button is displayed automatically that assists the user in selecting the invisible tabs.

Tabs in a TabPane may be positioned at the top, right, bottom, or left side of the TabPane. By default, they are positioned at the top.

Figure 12-71 shows two instances of a window. The window contains a TabPane with two tabs. In one instance, the General tab is selected, and in another, the Address tab is selected.



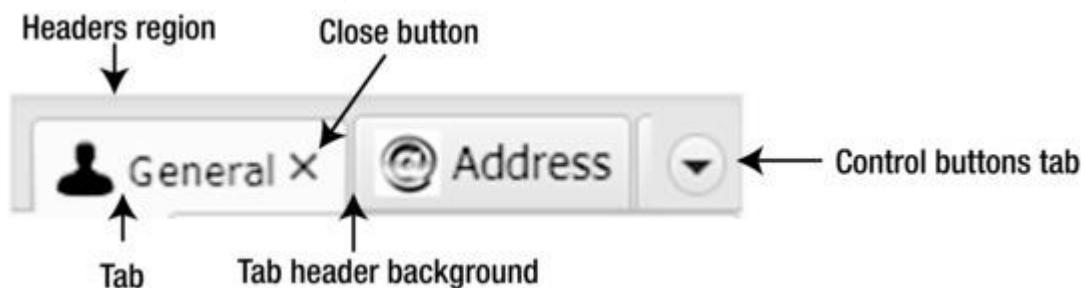
**Figure 12-71.** A window with a TabPane, which contains two tabs

A TabPane is divided into two parts: *header area* and *content area*. The header area displays the titles of tabs; the content area displays the content of the selected tab. The header area is subdivided into the following parts:

- Headers region
- Tab header background
- Control buttons tab
- Tab area

Figure 12-72 shows parts of the header area of a TabPane. The headers region is the entire header area. The tab header background is the area occupied by the titles of the tabs. The control buttons tab contains control buttons that are displayed when the width of the TabPane cannot display all of the tabs. The control button tab lets you select the tabs that are currently not visible. The tab area contains

a Label and a close button (the X icon next to the tab label). The Label displays the text and icon for a tab. The close button is used to close a selected tab.



**Figure 12-72.** Different parts of the header of a `TabPane`

## Creating Tabs

You can create a tab using the default constructor of the `Tab` class with an empty title:

```
Tab tab1 = new Tab();
```

Use the `setText()` method to set the title text for the tab:

```
tab1.setText("General");
```

The other constructor takes the title text as an argument:

```
Tab tab2 = new Tab("General");
```

## Setting the Title and Content of Tabs

The `Tab` class contains the following properties that let you set the title and content:

- `text`
- `graphic`
- `closable`
- `content`

The `text`, `graphic`, and `closable` properties specify what appears in the title bar of a tab. The `text` property specifies a string as the title text. The `graphic` property specifies a node as the title icon. Notice that the type of the `graphic` property is `Node`, so you can use any node as a graphic. Typically, a small icon is set as the graphic. The `text` property can be set in the constructor or using the `setText()` method. The following snippet of code creates a tab with text and sets an image as its graphic (assuming the file `resources/picture/address_icon.png` is included in the package):

```
// Create an ImageView for graphic
String imagePath = "resources/picture/address_icon.png";
```

```

URL imageUrl
= getClass().getClassLoader().getResource(imagePath);
Image img = new Image(imageUrl.toExternalForm());
ImageView icon = new ImageView(img);

// Create a Tab with "Address" text
Tab addressTab = new Tab("Address");

// Set the graphic
addressTab.setGraphic(icon);

```

The `closable` property is a boolean property that specifies whether the tab can be closed. If it is set to false, the tab cannot be closed. Closing of tabs is also controlled by the tab-closing policy of the `TabPane`. If the `closable` property is set to false, the tab cannot be closed by the user, irrespective of the tab-closing policy of the `TabPane`. You will learn about tab-closing policy when I discuss the `TabPane` later.

The `content` property is a node that specifies the content of the tab. The content of the tab is visible when the tab is selected. Typically, a layout pane with controls is set as the content of a tab. The following snippet of code creates a `GridPane`, adds some controls, and sets the `GridPane` as the content of a tab:

```

// Create a GridPane layout pane with some controls
GridPane grid = new GridPane();
grid.addRow(0, new Label("Street:"), streetFld);
grid.addRow(1, new Label("City:"), cityFld);
grid.addRow(2, new Label("State:"), stateFld);
grid.addRow(3, new Label("ZIP:"), zipFld);

Tab addressTab = new Tab("Address");
addressTab.setContent(grid); // Set the content

```

## Creating `TabPanes`

The `TabPane` class provides only one constructor—the default constructor. When you create a `TabPane`, it has no tabs:

```
TabPane tabPane = new TabPane();
```

## Adding Tabs to a `TabPane`

A `TabPane` stores the references of its tabs in an `ObservableList<Tab>`. The `getTabs()` method of the `TabPane` class returns the reference of the observable list. To add a tab to the `TabPane`, you need to add it to the observable list. The following snippet of code adds two tabs to a `TabPane`:

```

Tab generalTab = new Tab("General");
Tab addressTab = new Tab("Address");
...
TabPane tabPane = new TabPane();

```

```
// Add the two Tabs to the TabPane  
tabPane.getTabs().addAll(generalTab, addressTab);
```

When a tab is not supposed to be part of a TabPane, you need to remove it from the observable list. The TabPane will update its view automatically:

```
// Remove the Address tab  
tabPane.getTabs().remove(addressTab);
```

The read-only tabPane property of the Tab class stores the reference of the TabPane that contains the tab. If a tab has not yet been added to a TabPane, its tabPane property is null. Use the getTabPane() method of the Tab class to get the reference of the TabPane.

## Putting TabPanes and Tabs Together

I have covered enough information to allow you to see a TabPane with Tabs in action. Typically, a tab is reused. Inheriting a class from the Tab class helps when reusing a tab. Listing 12-41 and Listing 12-42 create two Tab classes. You will use them as tabs in subsequent examples. The GeneralTab class contains fields to enter the name and birth date of a person. The AddressTab class contains fields to enter an address.

### ***Listing 12-41.*** A GeneralTab Class that Inherits from the Tab Class

```
// GeneralTab.java  
package com.jdojo.control;  
  
import javafx.scene.Node;  
import javafx.scene.control.DatePicker;  
import javafx.scene.control.Label;  
import javafx.scene.control.Tab;  
import javafx.scene.control.TextField;  
import javafx.scene.layout.GridPane;  
  
public class GeneralTab extends Tab {  
    TextField firstNameFld = new TextField();  
    TextField lastNameFld = new TextField();  
    DatePicker dob = new DatePicker();  
  
    public GeneralTab(String text, Node graphic) {  
        this.setText(text);  
        this.setGraphic(graphic);  
        init();  
    }  
  
    public void init() {  
        dob.setPrefWidth(200);  
    }  
}
```

```

        GridPane grid = new GridPane();
        grid.addRow(0, new Label("First Name:"),  

firstNameFld);
        grid.addRow(1, new Label("Last Name:"),  

lastNameFld);
        grid.addRow(2, new Label("DOB:"), dob);
        this.setContent(grid);
    }
}

```

**Listing 12-42.** An AddressTab Class that Inherits from the Tab Class

```

// AddressTab.java
package com.jdojo.control;

import javafx.scene.Node;
import javafx.scene.control.Label;
import javafx.scene.control.Tab;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;

public class AddressTab extends Tab {
    TextField streetFld = new TextField();
    TextField cityFld = new TextField();
    TextField stateFld = new TextField();
    TextField zipFld = new TextField();

    public AddressTab(String text, Node graphic) {
        this.setText(text);
        this.setGraphic(graphic);
        init();
    }

    public void init() {
        GridPane grid = new GridPane();
        grid.addRow(0, new Label("Street:"), streetFld);
        grid.addRow(1, new Label("City:"), cityFld);
        grid.addRow(2, new Label("State:"), stateFld);
        grid.addRow(3, new Label("ZIP:"), zipFld);
        this.setContent(grid);
    }
}

```

The program in Listing 12-43 creates two tabs. They are instances of the GeneralTab and AddressTab classes. They are added to a TabPane, which is added to center region of a BorderPane. The program displays a window as shown in Figure 12-71.

**Listing 12-43.** Using a TabPane and Tabs Together

```

// TabTest.java
package com.jdojo.control;

```

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.TabPane;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class TabTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        ImageView privacyIcon
= getImage("privacy_icon.png");
        GeneralTab generalTab = new GeneralTab("General",
privacyIcon);

        ImageView addressIcon
= getImage("address_icon.png");
        AddressTab addressTab = new AddressTab("Address",
addressIcon);

        TabPane tabPane = new TabPane();
        tabPane.getTabs().addAll(generalTab, addressTab);

        BorderPane root = new BorderPane();
        root.setCenter(tabPane);
        root.setStyle("-fx-padding: 10;" +
"-fx-border-style: solid inside;" +
"-fx-border-width: 2;" +
"-fx-border-insets: 5;" +
"-fx-border-radius: 5;" +
"-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using TabPane and Tab Controls");
        stage.show();
    }

    public ImageView getImage(String fileName) {
        ImageView imgView = null;
        try {
            String imagePath = "resources/picture/"
+ fileName;
            Image img = new Image(imagePath);
            imgView = new ImageView(img);
        }
        catch(Exception e) {
            e.printStackTrace();
        }
        return imgView;
    }
}
```

```
    }  
}
```

## Understanding Tab Selection

TabPane supports single selection model, which allows selecting only one tab at a time. If a tab is selected by the user or programmatically, the previously selected tab is unselected. The Tab class provides the API to allow working with the selection state of an individual tab.

The TabPane class provides API that allows working with the selection of all of its tabs.

The Tab class contains a read-only selected property of the boolean type. It is true when the tab is selected. Otherwise, it is false. Note that it is a property of the Tab, not the TabPane.

Tab lets you add event handlers that are notified when the tab is selected or unselected. The onSelectionChanged property stores the reference of such an event:

```
Tab generalTab = ...  
generalTab.setOnSelectionChanged(e -> {  
    if (generalTab.isSelected()) {  
        System.out.println("General tab has been  
selected.");  
    } else {  
        System.out.println("General tab has been  
unselected.");  
    }  
});
```

TabPane tracks the selected tab and its index in the list of tabs. It uses a separate object, called *selection model*, for this purpose.

The TabPane class contains a selectionModel property to store the tab selection details. The property is an object of the SingleSelectionModel class. You can use your own selection model, which is almost never needed. The selection model provides the selection-related functionalities:

- It lets you select a tab using the index of the tab. The first tab has an index of 0.
- It lets you select the first, next, previous, or last tab in the list.
- It lets you clear the selection. Note that this feature is available, but is not commonly used. A TabPane should always typically have a selected tab.
- The selectedIndex and selectedItem properties track the index and reference of the selected tab. You can add

a ChangeListener to these properties to handle a change in tab selection in a TabPane.

By default, a TabPane selects its first tab. The following snippet of code selects the last Tab in a TabPane:

```
tabPane.getSelectionModel().selectLast();
```

Use the selectNext() method of the selection model to select the next tab from the list. Calling this method when the last tab is already selected has no effect.

Use the selectPrevious() and selectLast() methods to select the previous and the last tabs in the list. The select(int index) and select(T item) methods select a tab using the index and reference of the tab.

The program in Listing 12-44 adds two tabs to a TabPane. It adds a selection-changed event handler to both tabs. A ChangeListener is added to the selectedItem property of the selectionModel property of the TabPane. When a selection is made, a detailed message is printed on the standard output. Notice that a message is printed when you run the application because the TabPane selection model selects the first tab by default.

#### ***Listing 12-44.*** Tracking Tab Selection in a TabPane

```
// TabSelection.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.beans.value.ObservableValue;
import javafx.event.Event;
import javafx.scene.Scene;
import javafx.scene.control.Tab;
import javafx.scene.control.TabPane;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class TabSelection extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        GeneralTab generalTab = new GeneralTab("General",
null);
        AddressTab addressTab = new AddressTab("Address",
null);

        // Add selection a change listener to Tabs
```

```

        generalTab.setOnSelectionChanged(e ->
tabSelectedChanged(e));
        addressTab.setOnSelectionChanged(e ->
tabSelectedChanged(e));

        TabPane tabPane = new TabPane();

        // Add a ChangeListsner to the selection model
        tabPane.getSelectionModel().selectedItemProperty()
            .addListener(this::selectionChanged);

        tabPane.getTabs().addAll(generalTab, addressTab);

        HBox root = new HBox(tabPane);
        root.setStyle("-fx-padding: 10;" +
                      "-fx-border-style: solid inside;" +
                      "-fx-border-width: 2;" +
                      "-fx-border-insets: 5;" +
                      "-fx-border-radius: 5;" +
                      "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("TabPane Selection Model");
        stage.show();
    }

    public void selectionChanged(ObservableValue<? extends Tab>
prop,
                                Tab oldTab,
                                Tab newTab) {
        String oldTabText = oldTab == null? "None":oldTab.getText();
        String newTabText = newTab == null? "None":newTab.getText();
        System.out.println("Selection changed in TabPane:
old = " +
                           oldTabText + ", new = " + newTabText);
    }

    public void tabSelectedChanged(Event e) {
        Tab tab = (Tab)e.getSource();
        System.out.println("Selection changed event for "
+ tab.getText() +
                           " tab, selected = "
+ tab.isSelected());
    }
}

```

## Closing Tabs in a *TabPane*

Sometimes the user needs to add tabs to a *TabPane* on demand and they should be able to close tabs as well. For example, all modern web browsers use tabs for browsing and let you open and close tabs. Adding

tabs on demand requires some coding in JavaFX. However, closing tabs by the user is built in the `Tab` and `TabPane` classes.

Users can close Tabs in a `TabPane` using the close button that appears in the title bar of Tabs. The tab-closing feature is controlled by the following properties:

- The `closable` property of the `Tab` class
- The `tabClosingPolicy` property of the `TabPane` class

The `closable` property of a `Tab` class specifies whether the tab can be closed. If it is set to false, the tab cannot be closed, irrespective of the value for the `tabClosingPolicy`. The default value for the property is true. The `tabClosingPolicy` property specifies how the tab-closing buttons are available. Its value is one of the following constants of the `TabPane.TabClosingPolicy` enum:

- `ALL_TABS`
- `SELECTED_TAB`
- `UNAVAILABLE`

`ALL_TABS` means the close button is available for all tabs. That is, any tab can be closed at any time provided the `closable` property of the tab is true. `SELECTED_TAB` means the close button appears only for the selected tab. That is, only the selected tab can be closed at any time. This is the default tab-closing policy of a `TabPane`. `UNAVAILABLE` means the close button is not available for any tabs. That is, no tabs can be closed by the user, irrespective of their `closable` properties.

A distinction has to be made between:

- Closing tabs by the user using the close button
- Removing them programmatically by removing them from the observable list of `Tabs` of the `TabPane`

Both have the same effect, that `Tabs` are removed from the `TabPane`. The discussion in this section applies to closing tabs by the user.

The user action to closing tabs can be vetoed. You can add event handlers for the `TAB_CLOSE_REQUEST_EVENT` event for a tab. The event handler is called when the user attempts to close the tab. If the event handler consumes the event, the closing operation is canceled. You can use the `onCloseRequest` property of the `Tab` class to set such an event:

```
Tab myTab = new Tab("My Tab");
myTab.setOnCloseRequest(e -> { if (SOME_CONDITION_IS_TRUE) {
```

```

        // Cancel the close request
        e.consume();
    }
);
}

```

A tab also generates a closed event when it is closed by the user. Use the `onClosed` property of the `Tab` class to set a closed event handler for a tab. The event handler is typically used to release resources held by the tab:

```
myTab.setOnClosed(e -> {/* Release tab resources here */});
```

The program in Listing 12-45 shows how to use the tab-closing-related properties and events. It displays two tabs in a `TabPane`. A check box lets you veto the closing of tabs. Unless the check box is selected, an attempt to close tabs is vetoed on the close request event. If you close tabs, you can restore them using the Restore Tabs button. Use the tab-closing policy `ChoiceBox` to use a different tab-closing policy. For example, if you select `UNAVAILABLE` as the tab-closing policy, the close buttons will disappear from all tabs. When a tab is closed, a message is printed on the standard output.

### ***Listing 12-45.*** Using Properties and Events Related to Closing Tabs by Users

```

// TabClosingTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.beans.value.ObservableValue;
import javafx.collections.ObservableList;
import javafx.event.Event;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.CheckBox;
import javafx.scene.control.ChoiceBox;
import javafx.scene.control.Label;
import javafx.scene.control.Tab;
import javafx.scene.control.TabPane;
import javafx.stage.Stage;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import static javafx.scene.control.TabPane.TabClosingPolicy;

```

```

public class TabClosingTest extends Application {
    GeneralTab generalTab = new GeneralTab("General", null);
    AddressTab addressTab = new AddressTab("Address", null);
    TabPane tabPane = new TabPane();

    CheckBox allowClosingTabsFlag = new CheckBox("Are Tabs closable?");
    Button restoreTabsBtn = new Button("Restore Tabs");

```

```

ChoiceBox<TabPane.TabClosingPolicy> tabClosingPolicyChoices
= new ChoiceBox<>();

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

@Override
public void start(Stage stage) {
    // Add Tabs to the TabPane
    tabPane.getTabs().addAll(generalTab, addressTab);

    // Set a tab close request event handler for tabs
    generalTab.setOnCloseRequest(this::tabClosingRequest
ed);
    addressTab.setOnCloseRequest(this::tabClosingRequest
ed);

    // Set a closed event handler for the tabs
    generalTab.setOnClosed(e -> tabClosed(e));
    addressTab.setOnClosed(e -> tabClosed(e));

    // Set an action event handler for the restore
button
    restoreTabsBtn.setOnAction(e -> restoreTabs());

    // Add choices to the choice box
    tabClosingPolicyChoices.getItems()
        .addAll(TabClosingPolicy.ALL_TABS,
                TabClosingPolicy.SELECTED_TA
B,
                TabClosingPolicy.UNAVAILABLE
);

    // Set the default value for the tab closing policy
    tabClosingPolicyChoices.setValue(tabPane.getTabClosi
ngPolicy());
}

// Bind the tabClosingPolicy of the tabPane to the
value property of the
// of the ChoiceBoxx
tabPane.tabClosingPolicyProperty().bind(
    tabClosingPolicyChoices.val
ueProperty());

BorderPane root = new BorderPane();
GridPane grid = new GridPane();
grid.setHgap(10);
grid.setVgap(10);
grid.setStyle("-fx-padding: 10;");
grid.addRow(0, allowClosingTabsFlag,
restoreTabsBtn);
grid.addRow(1, new Label("Tab Closing Policy:"),
tabClosingPolicyChoices);

```

```

        root.setTop(grid);
        root.setCenter(tabPane);
        root.setStyle("-fx-padding: 10;" +
                      "-fx-border-style: solid inside;" +
                      "-fx-border-width: 2;" +
                      "-fx-border-insets: 5;" +
                      "-fx-border-radius: 5;" +
                      "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Closing Tabs");
        stage.show();
    }

    public void tabClosingRequested(Event e) {
        if (!allowClosingTabsFlag.isSelected()) {
            e.consume(); // Closing tabs is not allowed
        }
    }

    public void tabClosed(Event e) {
        Tab tab = (Tab)e.getSource();
        String text = tab.getText();
        System.out.println(text + " tab has been closed.");
    }

    public void restoreTabs() {
        ObservableList<Tab> list = tabPane.getTabs();
        if (!list.contains(generalTab)) {
            list.add(0, generalTab);
        }

        if (!list.contains(addressTab)) {
            list.add(1, addressTab);
        }
    }

    public void closingPolicyChanged(
        ObservableValue<? extends
TabPane.TabClosingPolicy> prop,
        TabPane.TabClosingPolicy oldPolicy,
        TabPane.TabClosingPolicy newPolicy) {
        tabPane.setTabClosingPolicy(newPolicy);
    }
}

```

## Positioning Tabs in a *TabPane*

Tabs in a *TabPane* may be positioned at the top, right, bottom, or left. The `side` property of the *TabPane* specifies the position of tabs. It is set to one of the constants of the `Side` enum:

- TOP
- RIGHT
- BOTTOM
- LEFT

The default value for the side property is `Side.TOP`. The following snippet of code creates a `TabPane` and sets the side property to `Side.LEFT` to position tabs on the left:

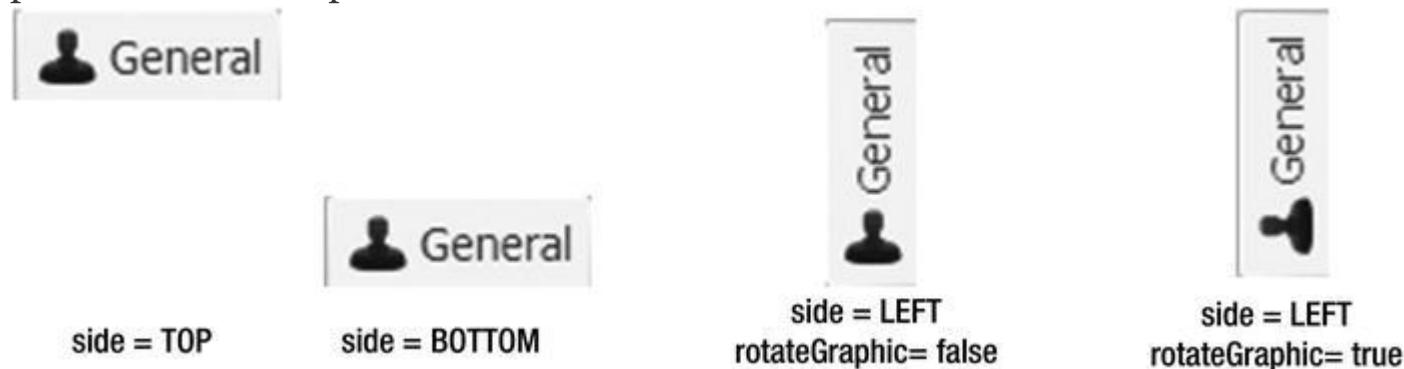
```
TabPane tabPane = new TabPane();
tabPane.setSide(Side.LEFT);
```

**Tip** The actual placement of tabs also uses the node orientation. For example, if the side property is set to `Side.LEFT` and the node orientation of the `TabPane` is set to `RIGHT_TO_LEFT`, the tabs will be positioned on the right side.

The `TabPane` class contains a `rotateGraphic` property, which is a boolean property. The property is related to the `side` property. When the `side` property is `Side.TOP` or `Side.BOTTOM`, the graphics of all tabs in their title bars are in the upright position. By default, when the `side` property changes to `Side.LEFT` or `Side.RIGHT`, the title text is rotated, keeping the graphic upright. The `rotateGraphic` property specifies whether the graphic is rotated with the text, as shown in the following code. By default, it is set to false.

```
// Rotate the graphic with the text for left and right sides
tabPane.setRotateGraphic(true);
```

Figure 12-73 shows the title bar of a tab in a `TabPane` with the `side` property set to `TOP` and `LEFT`. Notice the effect on the graphics when the `side` property is `LEFT` and the `rotateGraphic` property is false and true. The `rotateGraphic` property has no effect when tabs are positioned at the top or bottom.



**Figure 12-73.** Effects of the `side` and `rotateGraphic` properties of the `TabPane`

### Sizing Tabs in a `TabPane`

TabPane divides its layout into two parts:

- Header area
- Content area

The header area displays the titles of tabs. The content area displays the content of the selected tab. The size of the content area is automatically computed based on the content of all tabs. TabPane contains the following properties that allow you to set the minimum and maximum sizes of the title bars of tabs:

- tabMinHeight
- tabMaxHeight
- tabMinWidth
- tabMaxWidth

The default values are zero for minimum width and height, and Double.MAX\_VALUE for maximum width and height. The default size is computed based on the context of the tab titles. If you want all tab titles to be of a fixed size, set the minimum and maximum width and height to the same value. Note that for the fixed size tabs, the longer text in the title bar will be truncated.

The following snippet of code creates a TabPane and sets the properties, so all tabs are 100px wide and 30px tall:

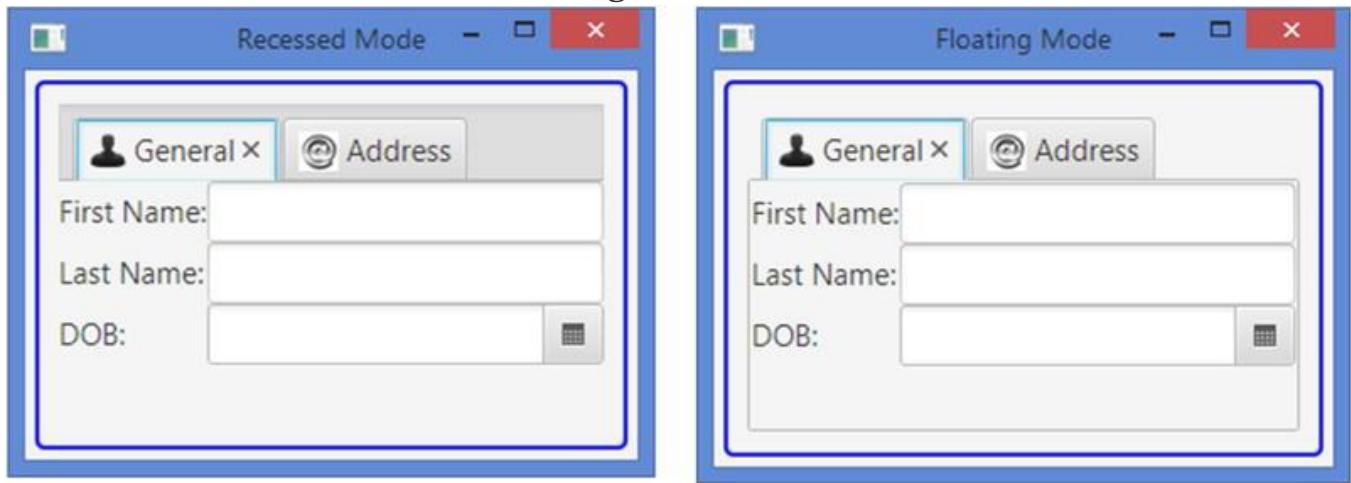
```
TabPane tabPane = new TabPane();
tabPane.setTabMinHeight(30);
tabPane.setTabMaxHeight(30);
tabPane.setTabMinWidth(100);
tabPane.setTabMaxWidth(100);
```

### Using Recessed and Floating TabPanes

A TabPane can be in recessed or floating mode. The default mode is recessed mode. In the recessed mode, it *appears* to be fixed. In floating mode, its appearance is changed to make it look like it is floating. In the floating mode, the background color of the header area is removed and a border around the content area is added. Here is a rule of thumb in deciding which mode to use:

- If you are using a TabPane along with other controls in a window, use floating mode.
- If the TabPane is the only one control on the window, use recessed mode.

Figure 12-74 shows two windows with the same TabPane: one in the recessed mode and one in the floating mode.



**Figure 12-74.** A TabPane in recessed and floating modes

The floating mode of a TabPane is specified by a style class. The TabPane class contains a `STYLE_CLASS_FLOATING` constant. If you add this style class to a TabPane, it is in the floating mode. Otherwise, it is in the recessed mode. The following snippet of code shows how to turn the floating mode for a TabPane on and off:

```
TabPane tabPane = new TabPane();

// Turn on the floating mode
tabPane.getStyleClass().add(TabPane.STYLE_CLASS_FLOATING);
...
// Turn off the floating mode
tabPane.getStyleClass().remove(TabPane.STYLE_CLASS_FLOATING);
```

### Styling Tab and TabPane with CSS

The default CSS style-class name for a tab and for a TabPane is `tab-pane`. You can style Tabs directly using the `tab` style class or using the substructure of TabPane. The later approach is commonly used. TabPane supports four CSS pseudo-classes, which correspond to the four values for its `side` property:

- top
- right
- bottom
- left

You can set the minimum and maximum sizes of the tab titles in a TabPane using the following CSS properties. They correspond to the

four properties in the `TabPane` class. Please refer to the “Sizing Tabs in a `TabPane`” section for a detailed discussion of these properties.

- `-fx-tab-min-width`
- `-fx-tab-max-width`
- `-fx-tab-min-height`
- `-fx-tab-max-height`

A `TabPane` divides its layout bounds into two areas: header area and content area. Please refer to Figure 12-72 for the different subparts in the header area. The header area is called the `tab-header-area` substructure, which contains the following substructures:

- `headers-region`
- `tab-header-background`
- `control-buttons-tab`
- `tab`

The `control-buttons-tab` substructure contains a `tab-down-button` substructure, which contains an `arrow` substructure.

The `tab` substructure contains `tab-label` and `tab-close-button` substructures. The `tab-content-area` substructure represents the content area of the `TabPane`. Substructures let you style different parts of `TabPane`.

The following code removes the background color for the header area as is done when the `TabPane` is in the floating mode:

```
.tab-pane > .tab-header-area > .tab-header-background {  
    -fx-background-color: null;  
}
```

The following code shows the text of the selected tab in boldface. Notice the use of the `selected` pseudo-class for the tab in the selector `.tab:selected`:

```
.tab-pane > .tab-header-area > .headers-region > .tab:selected >  
.tab-container > ,tab-label {  
    -fx-font-weight: bold;  
}
```

The following code shows Tabs in a `TabPane` in blue background with `10pt` white title text:

```
.tab-pane > .tab-header-area > .headers-region > .tab {  
    -fx-background-color: blue;  
}  
  
.tab-pane > .tab-header-area > .headers-region > .tab > .tab-  
container > .tab-label {  
    -fx-text-fill: white;
```

```

        -fx-font-size: 10pt;
    }

```

Use the `floating` style-class for the `TabPane` when styling it for the floating mode. The following style sets the border color to blue in floating mode:

```

.tab-pane.floating > .tab-content-area {
    -fx-border-color: blue;
}

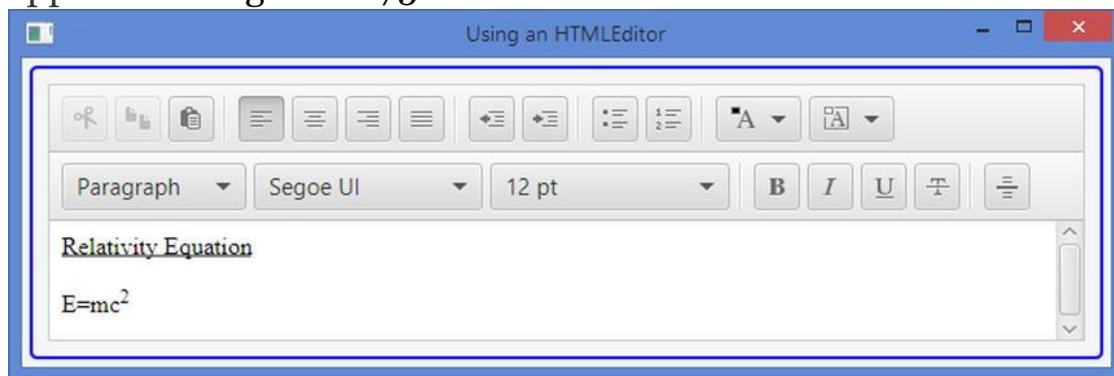
```

Please refer to the `modena.css` file for the complete list of styles used for `TabPane`.

## Understanding the `HTMLEditor` Control

The `HTMLEditor` control provides a rich text editing capability to JavaFX application. It uses HTML as its data model. That is, the formatted text in `HTMLEditor` is stored in HTML format.

An `HTMLEditor` control can be used for entering formatted text in a business application, for example, product description, or comments. It can also be used to enter e-mail content in an e-mail client application. Figure 12-75 shows a window with an `HTMLEditor` control.



**Figure 12-75.** An `HTMLEditor` control

An `HTMLEditor` displays formatting toolbars with it. You cannot hide the toolbars. They can be styled using a CSS. Using the toolbars, you can:

- Copy, cut, and paste text using the system clipboard
- Apply text alignment
- Indent text
- Apply bulleted list and numbered list styles
- Set foreground and background colors
- Apply paragraph and heading styles with font family and font size
- Apply formatting styles such as bold, italic, underline, and strikethrough
- Add horizontal rulers

The control supports HTML5. Note that the toolbars do not allow you to apply all kinds of HTML. However, if you load a document that uses those styles, it allows you to edit them. For example, you cannot create an HTML table directly in the control. However, if you load HTML content having HTML tables into the control, you will be able to edit the data in the tables.

The `HTMLEditor` does not provide API to load HTML content from a file to save its content to a file. You will have to write your own code to accomplish this.

### Creating an `HTMLEditor`

An instance of the `HTMLEditor` class represents an `HTMLEditor` control. The class is included in the `javafx.scene.web` package. Use the default constructor, which is the only constructor provided, to create an `HTMLEditor`:

```
HTMLEditor editor = new HTMLEditor();
```

### Using an `HTMLEditor`

The `HTMLEditor` class has a very simple API that consists of only three methods:

- `getHtmlText()`
- `setHtmlText(String htmlText)`
- `print(PrinterJob job)`

The `getHTMLText()` method returns the HTML content as a string. The `setHTMLText()` method sets the content of the control to the specified HTML string. The `print()` method prints the content of the control.

The program in Listing 12-46 shows how to use an `HTMLEditor`. It displays an `HTMLEditor`, a `TextArea`, and two `Buttons`. You can use the buttons to convert text in the `HTMLEditor` to HTML code and vice versa.

### ***Listing 12-46.*** Using the `HTMLEditor` Control

```
// HTMLEditorTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.TextArea;
```

```
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.web.HTMLEditor;
import javafx.stage.Stage;

public class HTMLEditorTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        HTMLEditor editor = new HTMLEditor();
        editor.setPrefSize(600, 300);

        TextArea html = new TextArea();
        html.setPrefSize(600, 300);
        html.setStyle("-fx-font-size:10pt; -fx-font-family:\\"Courier New\\\";");

        Button htmlToText = new Button("Convert HTML to
Text");
        Button textToHtml = new Button("Convert Text to
HTML");
        htmlToText.setOnAction(e ->
editor.setHtmlText(html.getText()));
        textToHtml.setOnAction(e ->
html.setText(editor.getHtmlText()));

        HBox buttons = new HBox(htmlToText, textToHtml);
        buttons.setSpacing(10);

        VBox root = new VBox(editor, buttons, html);
        root.setSpacing(10);
        root.setStyle("-fx-padding: 10;" +
"-fx-border-style: solid inside;" +
"-fx-border-width: 2;" +
"-fx-border-insets: 5;" +
"-fx-border-radius: 5;" +
"-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using an HTMLEditor");
        stage.show();
    }
}
```

## Styling *HTMLEditor* with CSS

The default CSS style-class name for an `HTMLEditor` is `html-editor`. The `HTMLEditor` uses styles of a `Control` such as padding, borders, and background color.

You can style each button in the toolbar separately. The following are the list of style-class names for the toolbar buttons. The names are self-explanatory, for example, `html-editor-align-right` and `html-editor-hr` are the style-class names for the toolbar buttons used to right align text and draw a horizontal ruler, respectively.

- `html-editor-cut`
- `html-editor-copy`
- `html-editor-paste`
- `html-editor-align-left`
- `html-editor-align-center`
- `html-editor-align-right`
- `html-editor-align-justify`
- `html-editor-outdent`
- `html-editor-indent`
- `html-editor-bullets`
- `html-editor-numbers`
- `html-editor-bold`
- `html-editor-italic`
- `html-editor-underline`
- `html-editor-strike`
- `html-editor-hr`

The following code sets a custom image for the Cut button in the toolbar:

```
.html-editor-cut {  
    -fx-graphic: url("my_html_editor_cut.jpg");  
}
```

Use the `button` and `toggle-button` style-class names if you want to apply styles to all toolbar buttons and toggle buttons:

```
/* Set the background colors for all buttons and toggle buttons */  
.html-editor .button, .html-editor .toggle-button {  
    -fx-background-color: lightblue;  
}
```

The HTMLEditor shows two ColorPickers for users to select the background and foreground colors. Their style-class names are `html-editor-background` and `html-editor-foreground`. The following code shows the selected color labels in the ColorPickers:

```
.html-editor-background {  
    -fx-color-label-visible: true;  
}  
  
.html-editor-foreground {  
    -fx-color-label-visible: true;  
}
```

## Choosing Files and Directories

JavaFX provides the `FileChooser` and `DirectoryChooser` classes in the `javafx.stage` package that are used to show file and directory dialogs. The dialogs have a platform dependent look and feel and cannot be styled using JavaFX. They are *not* controls. I am discussing them in this chapter because they are typically used along with controls. For example, a file or directory dialog is displayed when a button is clicked. On some platforms, for example, some mobile and embedded devices, users may have access to the file systems. Using these classes to access files and directories on such devices does nothing.

### The `FileChooser` Dialog

A `FileChooser` is a standard file dialog. It is used to let the user select files to open or save. Some of its parts, for example, the title, the initial directory, and the list of file extensions, can be specified before opening the dialogs. There are three steps in using a file dialog:

1. Create an object of the `FileChooser` class.
2. Set the initial properties for the file dialog.
3. Use one of the `showXXXDialog()` methods to show a specific type of file dialog.

### Creating a File Dialog

An instance of the `FileChooser` class is used to open file dialogs. The class contains a no-args constructor to create its objects:

```
// Create a file dialog  
FileChooser fileDialog = new FileChooser();
```

### Setting Initial Properties of the Dialog

You can set the following initial properties of the file dialog:

- Title
- `initialDirectory`
- `initialFileName`
- Extension filters

The `title` property of the `FileChooser` class is a string, which represents the title of the file dialog:

```
// Set the file dialog title  
fileDialog.setTitle("Open Resume");
```

The `initialDirectory` property of the `FileChooser` class is a `File`, which represents the initial directory when the file dialog is shown:

```
// Set C:\ as initial directory (on Windows)
fileDialog.setInitialDirectory(new File("C:\\\\"));
```

The `initialFileName` property of the `FileChooser` class is a string that is the initial file name for the file dialog. Typically, it is used for a file save dialog. Its effect depends on the platform if it is used for a file open dialog. For example, it is ignored on Windows:

```
// Set the initial file name
fileDialog.setInitialFileName("untitled.htm");
```

You can set a list of extension filters for a file dialog. Filters are displayed as a drop-down box. One filter is active at a time. The file dialog displays only those files that match the active extension filter. An extension filter is represented by an instance of the `ExtensionFilter` class, which is an inner static class of the `FileChooser` class.

The `getExtensionFilters()` method of the `FileChooser` class returns an `ObservableList<FileChooser.ExtensionFilter>`. You add the extension filters to the list. An extension filter has two properties: a description and a list of file extension in the form `*.<extension>`:

```
import static javafx.stage.FileChooser.ExtensionFilter.*;
...
// Add three extension filters
fileDialog.getExtensionFilters().addAll(
    new ExtensionFilter("HTML Files", "*htm", "*html"),
    new ExtensionFilter("Text Files", "*txt"),
    new ExtensionFilter("All Files", "*.*"));
```

By default, the first extension filter in the list is active when the file dialog is displayed. Use the `selectedExtensionFilter` property to specify the initial active filter when the file dialog is opened:

```
// Continuing with the above snippet of code, select *.txt filter
by default
fileDialog.setSelectedExtensionFilter(fileDialog.getExtensionFilters().get(1));
```

The same `selectedExtensionFilter` property contains the extension filter that is selected by the user when the file dialog is closed.

## Showing the Dialog

An instance of the `FileChooser` class can open three types of file dialogs:

- A file open dialog to select only one file
- A file open dialog to select multiple files
- A file save dialog

The following three methods of the `FileChooser` class are used to open three types of file dialogs:

- `showOpenDialog(Window ownerWindow)`
- `showOpenMultipleDialog(Window ownerWindow)`
- `showSaveDialog(Window ownerWindow)`

The methods do not return until the file dialog is closed. You can specify `null` as the owner window. If you specify an owner window, the input to the owner window is blocked when the file dialog is displayed.

The `showOpenDialog()` and `showSaveDialog()` methods return a `File` object, which is the selected file, or `null` if no file is selected.

The `showOpenMultipleDialog()` method returns a `List<File>`, which contains all selected files, or `null` if no files are selected:

```
// Show a file open dialog to select multiple files
List<File> files
= fileDialog.showOpenMultipleDialog(primaryStage);
if (files != null) {
    for(File f : files) {
        System.out.println("Selected file :" + f);
    }
} else {
    System.out.println("No files were selected.");
}
```

Use the `selectedExtensionFilter` property of the `FileChooser` class to get the selected extension filter at the time the file dialog was closed:

```
import static javafx.stage.FileChooser.ExtensionFilter;
...
// Print the selected extension filter description
ExtensionFilter filter = fileDialog.getSelectedExtensionFilter();
if (filter != null) {
    System.out.println("Selected Filter: "
+ filter.getDescription());
} else {
    System.out.println("No extension filter selected.");
}
```

## Using a File Dialog

The program in Listing 12-47 shows how to use open and save file dialogs. It displays a window with an `HTMLEditor` and three buttons. Use the Open button to open an HTML file in the editor. Edit the content in the editor. Use the Save button to save the content in the editor to a file. If you chose an existing file in the Save Resume dialog, the content of the file will be overwritten. It is left to the reader as an

exercise to enhance the program, so it will prompt the user before overwriting an existing file.

### ***Listing 12-47.*** Using Open and Save File Dialogs

```
// FileChooserTest.java
package com.jdojo.control;

import javafx.application.Application;
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.web.HTMLEditor;
import javafx.stage.FileChooser;
import javafx.stage.Stage;
import static javafx.stage.FileChooser.ExtensionFilter;

public class FileChooserTest extends Application {
    private Stage primaryStage;
    private HTMLEditor resumeEditor;
    private final FileChooser fileDialog = new FileChooser();

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

    @Override
    public void start(Stage stage) {
        primaryStage = stage; // Used in file dialogs later
        resumeEditor = new HTMLEditor();
        resumeEditor.setPrefSize(600, 300);

        // Filter only HTML files
        fileDialog.getExtensionFilters()
            .add(new ExtensionFilter("HTML Files",
                "*.htm", "*.html"));

        Button openBtn = new Button("Open");
        Button saveBtn = new Button("Save");
        Button closeBtn = new Button("Close");
        openBtn.setOnAction(e -> openFile());
        saveBtn.setOnAction(e -> saveFile());
        closeBtn.setOnAction(e -> stage.close());

        HBox buttons = new HBox(20, openBtn, saveBtn,
            closeBtn);
        buttons.setAlignment(Pos.CENTER_RIGHT);
        VBox root = new VBox(resumeEditor, buttons);
        root.setSpacing(20);
```

```

        root.setStyle("-fx-padding: 10;" +
                      "-fx-border-style: solid inside;" +
                      "-fx-border-width: 2;" +
                      "-fx-border-insets: 5;" +
                      "-fx-border-radius: 5;" +
                      "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Editing Resume in HTML Format");
        stage.show();
    }

    private void openFile() {
        fileDialog.setTitle("Open Resume");
        File file = fileDialog.showOpenDialog(primaryStage);
        if (file == null) {
            return;
        }

        try {
            // Read the file and populate the HTMLEditor
            byte[] resume
= Files.readAllBytes(file.toPath());
            resumeEditor.setHtmlText(new String(resume));
        }
        catch(IOException e) {
            e.printStackTrace();
        }
    }

    private void saveFile() {
        fileDialog.setTitle("Save Resume");
        fileDialog.setInitialFileName("untitled.htm");
        File file = fileDialog.showSaveDialog(primaryStage);
        if (file == null) {
            return;
        }

        try {
            // Write the HTML contents to the file.
Overwrite the existing file.
            String html = resumeEditor.getHtmlText();
            Files.write(file.toPath(), html.getBytes());
        }
        catch(IOException e) {
            e.printStackTrace();
        }
    }
}

```

## The *DirectoryChooser* Dialog

Sometimes you may need to let the user browse a directory from the available file systems on the computer. The `DirectoryChooser` class lets you display a platform-dependent directory dialog.

The `DirectoryChooser` class contains two properties:

- `title`
- `initialDirectory`

The `title` property is a string and it is the title of the directory dialog. The `initialDirectory` property is a `File` and it is the initial directory selected in the dialog when the dialog is shown.

Use the `showDialog(Window ownerWindow)` method of the `DirectoryChooser` class to open the directory dialog. When the dialog is opened, you can select at most one directory or close the dialog without selecting a directory. The method returns a `File`, which is the selected directory or `null` if no directory is selected. The method is blocked until the dialog is closed. If an owner window is specified, input to all windows in the owner window chain is blocked when the dialog is shown. You can specify a null owner window.

The following snippet of code shows how to create, configure, and display a directory dialog:

```
DirectoryChooser dirDialog = new DirectoryChooser();

// Configure the properties
dirDialog.setTitle("Select Destination Directory");
dirDialog.setInitialDirectory(new File("c:\\\\"));

// Show the directory dialog
File dir = dirDialog.showDialog(null);
if (dir != null) {
    System.out.println("Selected directory: " + dir);
} else {
    System.out.println("No directory was selected.");
}
```

## Summary

A user interface is a means to exchange information in terms of input and output between an application and its users. Entering text using a keyboard, selecting a menu item using a mouse, and clicking a button are examples of providing input to a GUI application. The application displays output on a computer monitor using text, charts, dialog boxes, among others. Users interact with a GUI application using graphical elements called *controls* or *widgets*. Buttons, labels, text fields, text area, radio buttons, and check boxes are a few examples of controls. JavaFX

provides a rich set of easy-to-use controls. Controls are added to layout panes that position and size them.

Each control in JavaFX is represented by an instance of a class. Control classes are included in the `javafx.scene.control` package. A control class in JavaFX is a subclass, direct or indirect, of the `Control` class, which in turn inherits from the `Region` class. Recall that the `Region` class inherits from the `Parent` class. Therefore, technically, a `Control` is also a `Parent`. A `Parent` can have children. However, control classes do not allow adding children. Typically, a control consists of multiple nodes that are internally maintained. Control classes expose the list of their internal unmodifiable children through the `getChildrenUnmodifiable()` method, which returns an `ObservableList<Node>`.

A labeled control contains a read-only textual content and optionally a graphic as part of its user interface. `Label`, `Button`, `CheckBox`, `RadioButton`, and `Hyperlink` are some examples of labeled controls in JavaFX. All labeled controls are inherited, directly or indirectly, from the `Labeled` class that, in turn, inherits from the `Control` class.

The `Labeled` class contains properties common to all labeled controls, such as content alignment, positioning of text relative to the graphic, and text font.

JavaFX provides button controls that can be used to execute commands, make choices, or both. All button control classes inherit from the `ButtonBase` class. All types of buttons support the `ActionEvent`. Buttons trigger an `ActionEvent` when they are activated. A button can be activated in different ways, for example, by using a mouse, a mnemonic, an accelerator key, or other key combinations. A button that executes a command when activated is known as a command button. The `Button`, `Hyperlink`, and `MenuItem` classes represent command buttons. A `MenuItem` lets the user execute a command from a list of commands. Buttons used for presenting different choices to users are known as choice buttons. The `ToggleButton`, `CheckBox`, and `RadioButton` classes represent choice buttons. The third kind of button is a hybrid of the first two kinds. They let users execute a command or make choices. The `SplitMenuItem` class represents a hybrid button.

JavaFX provides controls that let users select an item(s) from a list of items. They take less space compared to buttons. Those controls are `ChoiceBox`, `ComboBox`, `ListView`, `ColorPicker`, and `DatePicker`. `ChoiceBox` lets users select an item from a small list of predefined items. `ComboBox` is an advanced version

of `ChoiceBox`. It has many features, for example, an ability to be editable or changing the appearance of the items in the list, which are not offered in `ChoiceBox`. `ListView` provides users an ability to select multiple items from a list of items. Typically, all or more than one item in a `ListView` are visible to the user all of the time. `ColorPicker` lets users select a color from a standard color palette or define a custom color graphically. `DatePicker` lets users select a date from a calendar pop-up. Optionally, users can enter a date as text. `ComboBox`, `ColorPicker`, and `DatePicker` have the same superclass that is the `ComboBoxBase` class.

Text input controls let users work with single line or multiple lines of plain text. All text input controls are inherited from the `TextInputControl` class. There are three types of text input controls: `TextField`, `PasswordField`, and `TextArea`.

`TextField` lets the user enter a single line of plain text; newlines and tab characters in the text are removed. `PasswordField` inherits from `TextField`. It works much the same as `TextField`, except it masks its text. `TextArea` lets the user enter multiline plain text. A newline character starts a new paragraph in a `TextArea`.

For a long running task, you need to provide visual feedback to the user indicating the progress of the task for a better user experience.

The `ProgressIndicator` and `ProgressBar` controls are used to show the progress of a task. They differ in the ways they display the progress. The `ProgressBar` class inherits from the `ProgressIndicator` class. `ProgressIndicator` displays the progress in a circular control, whereas `ProgressBar` uses a horizontal bar.

`TitledPane` is a labeled control. It displays the text as its title. The graphic is shown in the title bar. Besides text and a graphic, it has content, which is a node. Typically, a group of controls is placed in a container and the container is added as the content for the `TitledPane`. `TitledPane` can be in a collapsed or expanded state. In the collapsed state, it displays only the title bar and hides the content. In the expanded state, it displays the title bar and the content. `Accordion` is a control that displays a group of `TitledPane` controls where only one of them is in the expanded state at a time.

`Pagination` is a control that is used to display a large single content by dividing it into smaller chunks called pages, for example, the results of a search.

A tool tip is a pop-up control used to show additional information about a node. It is displayed when a mouse pointer hovers over the node. There

is a small delay between when the mouse pointer hovers over a node and when the tool tip for the node is shown. The tool tip is hidden after a small period. It is also hidden when the mouse pointer leaves the control. You should not design a GUI application where the user depends on seeing tool tips for controls, as they may not be shown at all if the mouse pointer never hovers over the controls.

The `ScrollBar` and `ScrollPane` controls provide scrolling features to other controls. These controls are not used alone. They are always used to support scrolling in other controls.

Sometimes you want to place logically related controls side by side horizontally or vertically. For better appearance, controls are grouped using different types of separators.

The `Separator` and `SplitPane` controls are used for visually separating two controls or two groups of controls.

The `Slider` control lets the user select a numeric value from a numeric range graphically by sliding a thumb (or knob) along a track.

A `Slider` can be horizontal or vertical.

A menu is used to provide a list of actionable items to the user in a compact form. A menu bar is a horizontal bar that acts as a container for menus. An instance of the `MenuBar` class represents a menu bar.

A menu contains a list of actionable items, which are displayed on demand, for example, by clicking it. The list of menu items is hidden when the user selects an item or moves the mouse pointer outside the list. A menu is typically added to a menu bar or another menu as a submenu. An instance of the `Menu` class represents a menu.

A `Menu` displays text and a graphic. A menu item is an actionable item in a menu. The action associated with a menu item is performed by mouse or keys. Menu items can be styled using CSS. An instance of the `MenuItem` class represents a menu item. The `MenuItem` class is not a node. It is inherited from the `Object` class and, therefore, cannot be added directly to a scene graph. You need to add it to a `Menu`.

`ContextMenu` is a pop-up control that displays a list of menu items on request. It is known as a context or pop-up menu. By default, it is hidden. The user has to make a request, usually by right-clicking the mouse button, to show it. It is hidden once a selection is made. The user can dismiss a context menu by pressing the Esc key or clicking outside its bounds. An object of the `ContextMenu` class represents a context menu.

`ToolBar` is used to display a group of nodes, which provide the commonly used action items on a screen. Typically, a `ToolBar` contains the commonly used items that are also available through a menu and a context menu. A `ToolBar` can hold many types of nodes. The most commonly used nodes in a `ToolBar` are buttons and toggle

buttons. Separators are used to separate a group of buttons from others. Typically, buttons are kept smaller by using small icons, preferably 16px by 16px in size.

A window may not have enough space to display all of the pieces of information in a one-page view. TabPanes and Tabs let you present information in a page much better. A Tab represents a page and a TabPane contains the tabs. A Tab is not a control. An instance of the Tab class represents a Tab. The Tab class inherits from the Object class. However, a Tab supports some features as controls do, for example, they can be disabled, styled using CSS, and have context menus and tool tips.

A Tab consists of a title and content. The title consists of text, an optional graphic, and an optional close button to close the tab. The content consists of controls. Typically, the titles of tabs in a TabPane are visible. The content area is shared by all tabs. Tabs in a TabPane may be positioned at the top, right, bottom, or left side of the TabPane. By default, they are positioned at the top.

The HTMLEditor control provides a rich text editing capability to JavaFX application. It uses HTML as its data model. That is, the formatted text in HTMLEditor is stored in HTML format.

JavaFX provides the FileChooser and DirectoryChooser classes in the javafx.stage package that are used to show file and directory dialogs, respectively. The dialogs have a platform dependent look and feel and cannot be styled using JavaFX. They are not controls.

A FileChooser is a standard file dialog. It is used to let the user select files to open or save. A DirectoryChooser lets the user browse a directory from the available file systems on the machine.

The next chapter will discuss the TableView control that is used to display and edit data in tabular format.

## CHAPTER 13



### Understanding *TableView*

In this chapter, you will learn:

- What a TableView is
- How to create a TableView
- About adding columns to a TableView
- About populating a TableView with data
- About showing and hiding and reordering columns in a TableView
- About sorting and editing data in a TableView
- About adding and deleting rows in a TableView
- About resizing columns in a TableView
- About styling a TableView with CSS

### What Is a *TableView* ?

TableView is a powerful control to display and edit data in a tabular form from a data model. A TableView consists of rows and columns. A cell is an intersection of a row and a column. Cells contain the data values. Columns have headers that describe the type of data they contain. Columns can be nested. Resizing and sorting of column data have built-in support. Figure 13-1 shows a TableView with four columns that have the header text Id, First Name, Last Name, and Birth Date. It has five rows, with each row containing data for a person. For example, the cell in the fourth row and third column contains the last name Boyd.

<b>Id</b>	<b>First Name</b>	<b>Last Name</b>	<b>Birth Date</b>
1	Ashwin	Sharan	2012-10-11
2	Advik	Sharan	2012-10-11
3	Layne	Estes	2011-12-16
4	Mason	Boyd	2003-04-20
5	Babalu	Sharan	1980-01-10

**Figure 13-1.** A TableView showing a list of persons

TableView is a powerful, but not simple, control. You need to write a few lines of code to use even the simplest TableView that displays some meaningful data to users. There are several classes involved in working with TableView. I will discuss these classes in detail when I discuss the different features of the TableView:

- TableView
- TableColumn
- TableRow
- TableCell
- TablePosition
- TableView.TableViewFocusModel
- TableView.TableViewSelectionModel

The TableView class represents a TableView control.

The TableColumn class represents a column in a TableView.

Typically, a TableView contains multiple instances of TableColumn.

A TableColumn consists of cells, which are instances of the TableCell class. A TableColumn uses two properties to populate cells and render values in them. It uses a cell value factory to extract the value for its cells from the list of items. It uses a cell factory to render data in a cell. You must specify a cell value factory for a TableColumn to see some data in it. A TableColumn uses a default cell factory that knows how to render text and a graphic node.

The TableRow class inherits from the IndexedCell class. An instance of TableRow represents a row in a TableView. You would almost never use this class in an application unless you want to provide a customized implementation for rows. Typically, you customize cells, not rows.

An instance of the TableCell class represents a cell in a TableView. Cells are highly customizable. They display data from the underlying data model for the TableView. They are capable of displaying data as well as graphics.

The TableColumn, TableRow, and TableCell classes contain a tableView property that holds the reference of the TableView that contains them. The tableView property contains null when the TableColumn does not belong to a TableView.

A TablePosition represents the position of a cell.

Its getRow() and getColumn() methods return the indices of the row and column, respectively, to which the cell belongs.

The TableViewFocusModel class is an inner static class of the TableView class. It represents the focus model for the TableView to manage focus for rows and cells.

The TableViewSelectionModel class is an inner static class of the TableView class. It represents the selection model for the TableView to manage selection for rows and cells.

Like ListView and TreeView controls, TableView is virtualized. It creates just enough cells to display the visible content. As you scroll through the content, the cells are recycled. This helps keep the number of nodes in the scene graph to a minimum. Suppose you have ten columns and 1,000 rows in a TableView and only ten rows are visible at a time. An inefficient approach would be to create 10,000 cells, one cell for each piece of data. The TableView creates only 100 cells, so it can display ten rows with ten columns. As you scroll through the content, the same 100 cells will be recycled to show the other visible rows. Virtualization makes it possible to use TableView with a large data model without performance penalty for viewing the data in a chunk.

For examples in this chapter, I will use the Person class from Chapter 11 on MVC. The Person class is in the com.jdojo.mvc.model package. Before I start discussing the TableView control in detail, I will introduce a PersonTableUtil class, as shown in Listing 13-1. I will reuse it several times in the examples presented. It has static methods to return an observable list of persona and instances of the TableColumn class to represent columns in a TableView.

### ***Listing 13-1. A PersonTableUtil Utility Class***

```
// PersonTableUtil.java
package com.jdojo.control;

import com.jdojo.mvc.model.Person;
import java.time.LocalDate;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.control.TableColumn;
import javafx.scene.control.cell.PropertyValueFactory;

public class PersonTableUtil {
    /* Returns an observable list of persons */
    public static ObservableList<Person> getPersonList() {
        Person p1 = new Person("Ashwin", "Sharan",
LocalDate.of(2012, 10, 11));
        Person p2 = new Person("Advik", "Sharan",
LocalDate.of(2012, 10, 11));
        Person p3 = new Person("Layne", "Estes",

```

```

LocalDate.of(2011, 12, 16));
        Person p4 = new Person("Mason", "Boyd",
LocalDate.of(2003, 4, 20));
        Person p5 = new Person("Babalu", "Sharan",
LocalDate.of(1980, 1, 10));
        return FXCollections.<Person>observableArrayList(p1,
p2, p3, p4, p5);
    }

    /* Returns Person Id TableColumn */
    public static TableColumn<Person, Integer> getIdColumn() {
        TableColumn<Person, Integer> personIdCol = new
TableColumn<>("Id");
        personIdCol.setCellValueFactory(new
PropertyValueFactory<>("personId"));
        return personIdCol;
    }

    /* Returns First Name TableColumn */
    public static TableColumn<Person, String>
getFirstNameColumn() {
        TableColumn<Person, String> fNameCol = new
TableColumn<>("First Name");
        fNameCol.setCellValueFactory(new
PropertyValueFactory<>("firstName"));
        return fNameCol;
    }

    /* Returns Last Name TableColumn */
    public static TableColumn<Person, String>
getLastColumnName() {
        TableColumn<Person, String> lastNameCol = new
TableColumn<>("Last Name");
        lastNameCol.setCellValueFactory(new
PropertyValueFactory<>("lastName"));
        return lastNameCol;
    }

    /* Returns Birth Date TableColumn */
    public static TableColumn<Person, LocalDate>
getBirthDateColumn() {
        TableColumn<Person, LocalDate> bDateCol =
                new TableColumn<>("Birth Date");
        bDateCol.setCellValueFactory(new
PropertyValueFactory<>("birthDate"));
        return bDateCol;
    }
}

```

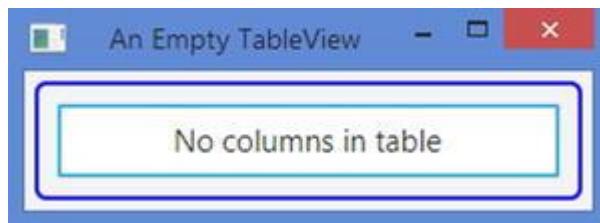
Subsequent sections will walk you through the steps to display and edit data in a *TableView*.

## Creating a *TableView*

In the following example, you will use the `TableView` class to create a `TableView` control. `TableView` is a parameterized class, which takes the type of items the `TableView` contains. Optionally, you can pass the model into its constructor that supplies the data. The constructor creates a `TableView` without a model. The following statement creates a `TableView` that will use objects of the `Person` class as its items:

```
TableView<Person> table = new TableView<>();
```

When you add the above `TableView` to a scene, it displays a placeholder, as shown in Figure 13-2. The placeholder lets you know that you need to add columns to the `TableView`. There must be at least one visible leaf column in the `TableView` data.



**Figure 13-2.** A `TableView` with no columns and data showing a placeholder

You would use another constructor of the `TableView` class to specify the model. It accepts an observable list of items. The following statement passes an observable list of `Person` objects as the initial data for the `TableView`:

```
TableView<Person> table = new  
TableView<>(PersonTableUtil.getPersonList());
```

### Adding Columns to a `TableView`

An instance of the  `TableColumn` class represents a column in a `TableView`. A  `TableColumn` is responsible for displaying and editing the data in its cells. A  `TableColumn` has a header that can display header text, a graphic, or both. You can have a context menu for a  `TableColumn`, which is displayed when the user right-clicks inside the column header. Use the `contextMenu` property to set a context menu.

The  `TableColumn<S, T>` class is a generic class. The `S` parameter is the items type, which is of the same type as the parameter of the `TableView`. The `T` parameter is the type of data in all cells of the column. For example, an instance of the  `TableColumn<Person, Integer>` may be used to represent a column to display the ID of a Person, which is of `int` type; an instance of the  `TableColumn<Person, String>` may be used to represent a column to display the *first name* of a person, which is of `String` type.

The following snippet of code creates a `TableColumn` with First Name as its header text:

```
 TableColumn<Person, String> fNameCol = new TableColumn<>("First Name");
```

A `TableColumn` needs to know how to get the value (or data) for its cells from the model. To populate the cells, you need to set the `cellValueFactory` property of the `TableColumn`. If the model for a `TableView` contains objects of a class that is based on JavaFX properties, you can use an object of the `PropertyValueFactory` class as the cell value factory, which takes the property name. It reads the property value from the model and populates all of the cells in the column, as in the following code:

```
// Use the firstName property of Person object to populate the column cells
PropertyValueFactory<Person, String> fNameCellValueFactory =
    new PropertyValueFactory<>("firstName");
fNameCol.setCellValueFactory(fNameCellValueFactory);
```

You need to create a `TableColumn` object for each column in the `TableView` and set its cell value factory property. The next section will explain what to do if your item class is not based on JavaFX properties or you want to populate the cells with computed values.

The last step in setting up a `TableView` is to add `TableColumns` to its list of columns. A `TableView` stores references of its columns in an `ObservableList<TableColumn>` whose reference can be obtained using the `getColumns()` method of the `TableView`:

```
// Add the First Name column to the TableView
table.getColumns().add(fNameCol);
```

That is all it takes to use a `TableView` in its simplest form, which is not so “simple” after all! The program in Listing 13-2 shows how to create a `TableView` with a model and add columns to it. It uses the `PersonTableUtil` class to get the list of persons and columns. The program displays a window as shown in Figure 13-3.

### ***Listing 13-2.*** Using `TableView` in Its Simplest Form

```
// SimplestTableView.java
package com.jdojo.control;

import com.jdojo.mvc.model.Person;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.TableView;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class SimplestTableView extends Application {
```

```

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

@Override
public void start(Stage stage) {
    // Create a TableView with a list of persons
    TableView<Person> table = new
    TableView<>(PersonTableUtil.getPersonList());

    // Add columns to the TableView
    table.getColumns().addAll(PersonTableUtil.getIdColumn(),
        PersonTableUtil.getFirstNameColumn(),
        PersonTableUtil.getLastNameColumn(),
        PersonTableUtil.getBirthDateColumn());

    VBox root = new VBox(table);
    root.setStyle("-fx-padding: 10;" +
        "-fx-border-style: solid inside;" +
        "-fx-border-width: 2;" +
        "-fx-border-insets: 5;" +
        "-fx-border-radius: 5;" +
        "-fx-border-color: blue;");

    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("Simplest TableView");
    stage.show();
}
}

```



**Figure 13-3.** A window with a `TableView` that displays four columns and five rows

`TableView` supports nesting of columns. For example, you can have two columns, First and Last, nested inside a Name column.

A TableColumn stores the list of nested columns in an observable list whose reference can be obtained using the getColumns() method of the TableColumn class. The innermost nested columns are known as *leaf columns*. You need to add the cell value factories for the leaf columns. Nested columns only provide visual effects. The following snippet of code creates a TableView and adds an Id column and two leaf columns, First and Last, that are nested in the Name column. The resulting TableView is shown in Figure 13-4. Note that you add the topmost columns to the TableView, not the nested columns. TableView takes care of adding all nested columns for the topmost columns. There is no limit on the level of column nesting.

```
// Create a TableView with data
TableView<Person> table = new
TableView<>(PersonTableUtil.getPersonList());

// Create leaf columns - Id, First and Last
TableColumn<Person, String> idCol = new TableColumn<>("Id");
idCol.setCellValueFactory(new
PropertyValueFactory<>("personId"));

TableColumn<Person, String> fNameCol = new
TableColumn<>("First");
fNameCol.setCellValueFactory(new
PropertyValueFactory<>("firstName"));

TableColumn<Person, String> lNameCol = new TableColumn<>("Last");
lNameCol.setCellValueFactory(new
PropertyValueFactory<>("lastName"));

// Create Name column and nest First and Last columns in it
TableColumn<Person, String> nameCol = new TableColumn<>("Name");
nameCol.getColumns().addAll(fNameCol, lNameCol);

// Add columns to the TableView
table.getColumns().addAll(idCol, nameCol);
```

Id	Name	
	First	Last
1	Ashwin	Sharan
2	Advik	Sharan
3	Layne	Estes
4	Mason	Boyd
5	Babalu	Sharan

**Figure 13-4.** A TableView with nested columns

The following methods in the `TableView` class provide information about visible leaf columns:

```
 TableColumn<S,?> getVisibleLeafColumn(int columnIndex)
 ObservableList<TableColumn<S,?>> getVisibleLeafColumns()
 int getVisibleLeafIndex(TableColumn<S,?> column)
```

The `getVisibleLeafColumn()` method returns the reference of the column for the specified column index. The column index is counted only for visible leaf column and the index starts at zero.

The `getVisibleLeafColumns()` method returns an observable list of all visible leaf columns. The `getVisibleLeafIndex()` method returns the column reference for the specified column index of a visible leaf column.

### Customizing TableView Placeholder

`TableView` displays a placeholder when it does not have any visible leaf columns or content. Consider the following snippet of code that creates a `TableView` and adds columns to it:

```
 TableView<Person> table = new TableView<>();
 table.getColumns().addAll(PersonTableUtil.getIdColumn(),
                           PersonTableUtil.getFirstNameColumn(),
                           PersonTableUtil.getLastNameColumn(),
                           PersonTableUtil.getBirthDateColumn());
```

Figure 13-5 shows the results of the above `TableView`. Columns and a placeholder are displayed, indicating that the `TableView` does not have data.

<b>Id</b>	<b>First Name</b>	<b>Last Name</b>	<b>Birth Date</b>
No content in table			

**Figure 13-5.** A `TableView` control with columns and no data

You can replace the built-in placeholder using the `placeholder` property of the `TableView`. The value for the property is an instance of the `Node` class. The following statement sets a `Label` with a generic message as a placeholder:

```
table.setPlaceholder(new Label("No visible columns and/or data exist."));
```

You can set a custom placeholder to inform the user of the specific condition that resulted in showing no data in the `TableView`. The following statement uses binding to change the placeholder as the conditions change:

```

table.placeholderProperty().bind(
    new When(new SimpleIntegerProperty(0)
        .isEqualTo(table.getVisibleLeafColumns().size()))
)
    .then(new When(new SimpleIntegerProperty(0)
        .isEqualTo(table.getItems().size())))
        .then(new Label("No columns and data
exist."))
        .otherwise(new Label("No columns exist."))
.otherwise(new When(new SimpleIntegerProperty(0)
    .isEqualTo(table.getItems().size()))
    .then(new Label("No data exist."))
    .otherwise((Label)null));

```

## Populating a *TableColumn* with Data

Cells in a row of a `TableView` contain data related to an item such as a person, a book, and so forth. Data for some cells in a row may come directly from the attributes of the item or they may be computed.

`TableView` has an `items` property of the `ObservableList<S>` type. The generic type `S` is the same as the generic type of the `TableView`. It is the data model for the `TableView`. Each element in the `items` list represents a row in the `TableView`. Adding a new item to the `items` list adds a new row to the `TableView`. Deleting an item from the `items` list deletes the corresponding row from the `TableView`.

**Tip** Whether updating an item in the `items` list updates the corresponding data in the `TableView` depends on how the cell value factory for the column is set up. I will discuss examples of both kinds in this section.

The following snippet of code creates a `TableView` in which a row represents a `Person` object. It adds data for two rows:

```

TableView<Person> table = new TableView<>();

Person p1 = new Person("John", "Jacobs", null);
Person p2 = new Person("Donna", "Duncan", null);
table.getItems().addAll(p1, p2);

```

Adding items to a `TableView` is useless unless you add columns to it. Among several other things, a `TableColumn` object defines:

- Header text and graphic for the column
- A cell value factory to populate the cells in the column

The `TableColumn` class gives you full control over how cells in a column are populated. The `cellValueFactory` property of the `TableColumn` class is responsible for populating cells of the column.

A cell value factory is an object of the `Callback` class, which receives a  `TableColumn.CellDataFeatures`  object and returns an  `ObservableValue` .

The  `CellDataFeatures`  class is a static inner class of the  `TableColumn`  class, which wraps the reference of the  `TableView` ,  `TableColumn` , and the item for the row for which the cells of the column are being populated. Use the  `getTableView()` ,  `getColumn()` , and  `getValue()`  methods of the  `CellDataFeatures`  class to get the reference of the  `TableView` ,  `TableColumn` , and the item for the row, respectively.

When the  `TableView`  needs the value for a cell, it calls the  `call()`  method of the cell value factory object of the column to which the cell belongs. The  `call()`  method is supposed to return the reference of an  `ObservableValue`  object, which is monitored for any changes. The return  `ObservableValue`  object may contain any type of object. If it contains a node, the node is displayed as a graphic in the cell. Otherwise, the  `toString()`  method of the object is called and the returned string is displayed in the cell.

The following snippet of code creates a cell value factory using an anonymous class. The factory returns the reference of the  `firstName`  property of the  `Person`  class. Note that a JavaFX property is an  `ObservableValue` .

```
import static javafx.scene.control.TableColumn.CellDataFeatures;
...
// Create a String column with the header "First Name" for Person
object
TableColumn<Person, String> fNameCol = new TableColumn<>("First
Name");

// Create a cell value factory object
Callback<CellDataFeatures<Person, String>,
ObservableValue<String>> fNameCellFactory =
new Callback<CellDataFeatures<Person, String>,
ObservableValue<String>>() {
@Override
public ObservableValue<String> call(CellDataFeatures<Person,
String> cellData) {
    Person p = cellData.getValue();
    return p.firstNameProperty();
}};

// Set the cell value factory
fNameCol.setCellValueFactory(fNameCellFactory);
```

Using a lambda expression to create and set a cell value factory comes in handy. The above snippet of code can be written as follows:

```
TableColumn<Person, String> fNameCol = new TableColumn<>("First Name");
fNameCol.setCellValueFactory(cellData ->
cellData.getValue().firstNameProperty());
```

When a JavaFX property supplies values for cells in a column, creating the cell value factory is easier if you use an object of the PropertyValueFactory class. You need to pass the name of the JavaFX property to its constructor. The following snippet of code does the same as the code shown above. You would take this approach to create TableColumn objects inside the utility methods in the PersonTableUtil class.

```
TableColumn<Person, String> fNameCol = new TableColumn<>("First Name");
fNameCol.setCellValueFactory(new PropertyValueFactory<>("firstName"));
```

**Tip** Using JavaFX properties as the value supplied for cells has a big advantage. The TableView keeps the value in the property and the cell in sync. Changing the property value in the model automatically updates the value in the cell.

TableColumn also supports POJO (Plain Old Java Object) as items in the TableView. The disadvantage is that when the model is updated, the cell values are not automatically updated. You use the same PropertyValueFactory class to create the cell value factory. The class will look for the public getter and setter methods with the property name you pass. If only the getter method is found, the cell will be read-only. For an `xxx` property, it tries looking for `getXXX()` and `setXXX()` methods using the JavaBeans naming conventions. If the type of `xxx` is boolean, it also looks for the `isXXX()` method. If a getter or a setter method is not found, a runtime exception is thrown. The following snippet of code creates a column with the header text Age Category:

```
TableColumn<Person, Person.AgeCategory> ageCategoryCol =
    new TableColumn<>("Age Category");
ageCategoryCol.setCellValueFactory(new PropertyValueFactory<>("ageCategory"));
```

It indicates that the items type is Person and the column type is Person.AgeCategory. It passes ageCategory as the property name into the constructor of the PropertyValueFactory class. First, the class will look for an ageCategory property in the Person class. The Person class does not have this property. Therefore, it will try using Person class as a POJO for this property. Then it will look for `getAgeCategory()` and `setAgeCategory()` methods in

the Person class. It finds only the getter method, `getAgeCategory()`, and hence, it will make the column read-only.

The values in the cells of a column do not necessarily have to come from JavaFX or POJO properties. They can be computed using some logic. In such cases, you need to create a custom cell value factory and return a `ReadOnlyXxxWrapper` object that wraps the computed value. The following snippet of code creates an Age column that displays a computed age in years:

```
TableColumn<Person, String> ageCol = new TableColumn<>("Age");
ageCol.setCellValueFactory(cellData -> {
    Person p = cellData.getValue();
    LocalDate dob = p.getBirthDate();
    String ageInYear = "Unknown";
    if (dob != null) {
        long years = YEARS.between(dob, LocalDate.now());
        if (years == 0) {
            ageInYear = "< 1 year";
        } else if (years == 1) {
            ageInYear = years + " year";
        } else {
            ageInYear = years + " years";
        }
    }
    return new ReadOnlyStringWrapper(ageInYear);
});
```

This completes the different ways of setting the cell value factory for cells of a column in a `TableView`. The program in Listing 13-3 creates cell value factories for JavaFX properties, a POJO property, and a computed value. It displays a window as shown in Figure 13-6.

### ***Listing 13-3.*** Setting Cell Value Factories for Columns

```
// TableViewDataTest.java
package com.jdojo.control;

import com.jdojo.mvc.model.Person;
import javafx.application.Application;
import javafx.beans.property.ReadOnlyStringWrapper;
import javafx.scene.Scene;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.cell.PropertyValueFactory;
import java.time.LocalDate;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;
import static java.time.temporal.ChronoUnit.YEARS;

public class TableViewDataTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
```

```

    }

@Override
@SuppressWarnings("unchecked")
public void start(Stage stage) {
    // Create a TableView with data
    TableView<Person> table =
        new
    TableView<>(PersonTableUtil.getPersonList());

    // Create an "Age" computed column
    TableColumn<Person, String> ageCol = new
    TableColumn<>("Age");
    ageCol.setCellValueFactory(cellData -> {
        Person p = cellData.getValue();
        LocalDate dob = p.getBirthDate();
        String ageInYear = "Unknown";

        if (dob != null) {
            long years = YEARS.between(dob,
LocalDate.now());
            if (years == 0) {
                ageInYear = "< 1 year";
            } else if (years == 1) {
                ageInYear = years + "
year";
            } else {
                ageInYear = years + "
years";
            }
        }
        return new ReadOnlyStringWrapper(ageInYear);
    });

    // Create an "Age Category" column
    TableColumn<Person, Person.AgeCategory>
ageCategoryCol =
        new TableColumn<>("Age Category");
    ageCategoryCol.setCellValueFactory(
        new
    PropertyValueFactory<>("ageCategory"));

    // Add columns to the TableView
    table.getColumns().addAll(PersonTableUtil.getIdColum
n(),
        PersonTableUtil.getFirstNameColu
mn(),
        PersonTableUtil.getLastNameColu
n(),
        PersonTableUtil.getBirthDateColu
mn(),
        ageCol,
        ageCategoryCol);
}

```

```

HBox root = new HBox(table);
root.setStyle("-fx-padding: 10;" +
              "-fx-border-style: solid inside;" +
              "-fx-border-width: 2;" +
              "-fx-border-insets: 5;" +
              "-fx-border-radius: 5;" +
              "-fx-border-color: blue;");

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("Populating TableViews");
stage.show();
}
}

```

Id	First Name	Last Name	Birth Date	Age	Age Category
1	Ashwin	Sharan	2012-10-11	2 years	CHILD
2	Advik	Sharan	2012-10-11	2 years	CHILD
3	Layne	Estes	2011-12-16	3 years	CHILD
4	Mason	Boyd	2003-04-20	11 years	CHILD
5	Babalu	Sharan	1980-01-10	35 years	ADULT

**Figure 13-6.** A TableView having columns for JavaFX properties, POJO properties, and computed values

Cells in a TableView can display text and graphics. If the cell value factory returns an instance of the Node class, which could be an ImageView, the cell displays it as graphic. Otherwise, it displays the string returned from the `toString()` method of the object. It is possible to display other controls and containers in cells. However, a TableView is not meant for that and such uses are discouraged. Sometimes using a specific type of control in a cell, for example, a check box, to show or edit a boolean value provides a better user experience. I will cover such customization of cells shortly.

## Using a Map as Items in a TableView

Sometimes data in a row for a TableView may not map to a domain object, for example, you may want to display the result set of a dynamic

query in a TableView. The items list consists of an observable list of Map. A Map in the list contains values for all columns in the row. You can define a custom cell value factory to extract the data from the Map. The MapValueFactory class is especially designed for this purpose. It is an implementation of the cell value factory, which reads data from a Map for a specified key.

The following snippet of code creates a TableView of Map. It creates an Id column and sets an instance of the MapValueFactory class as its cell value factory specifying the idColumnKey as the key that contains the value for the Id column. It creates a Map and populates the Id column using the idColumnKey. You need to repeat these steps for all columns and rows.

```
TableView<Map> table = new TableView<>();

// Define the column, its cell value factory and add it to the
// TableView
String idColumnKey = "id";
TableColumn<Map, Integer> idCol = new TableColumn<>("Id");
idCol.setCellValueFactory(new MapValueFactory<>(idColumnKey));
table.getColumns().add(idCol);

// Create and populate a Map an item
Map row1 = new HashMap();
row1.put(idColumnKey, 1);

// Add the Map to the TableView items list
table.getItems().add(row1);
```

The program in Listing 13-4 shows how to use the MapValueFactory as the cell value factory for columns in a TableView. It displays the person's data returned by the getPersonList() method in the PersonTableUtil class.

### ***Listing 13-4. Using MapValueFactory as a Cell Value Factory for Cells in a TableView***

```
// TableViewMapDataTest.java
package com.jdojo.control;

import com.jdojo.mvc.model.Person;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.TableColumn;
import javafx.scene.control TableView;
import java.time.LocalDate;
import java.util.HashMap;
import java.util.Map;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
```

```

import javafx.scene.control.cell.MapValueFactory;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class TableViewMapDataTest extends Application {
    private final String idColumnKey = "id";
    private final String firstNameColumnKey = "firstName";
    private final String lastNameColumnKey = "lastName";
    private final String birthDateColumnKey = "birthDate";

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

    @Override
    public void start(Stage stage) {
        TableView<Map> table = new TableView<>();
        ObservableList<Map<String, Object>> items
= this.getMapData();
        table.getItems().addAll(items);
        this.addColumns(table);

        HBox root = new HBox(table);
        root.setStyle("-fx-padding: 10;" +
                     "-fx-border-style: solid inside;" +
                     "-fx-border-width: 2;" +
                     "-fx-border-insets: 5;" +
                     "-fx-border-radius: 5;" +
                     "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using a Map as items in
a TableView");
        stage.show();
    }

    public ObservableList<Map<String, Object>> getMapData() {
        ObservableList<Map<String, Object>> items =
            FXCollections.<Map<String,
Object>>observableArrayList();

        // Extract the person data, add the data to a Map,
and add the Map to
        // the items list
        ObservableList<Person> persons
= PersonTableUtil.getPersonList();
        for(Person p : persons) {
            Map<String, Object> map = new HashMap<>();
            map.put(idColumnKey, p.getPersonId());
            map.put(firstNameColumnKey, p.getFirstName());
            map.put(lastNameColumnKey, p.getLastName());
            map.put(birthDateColumnKey, p.getBirthDate());
            items.add(map);
        }
    }
}

```

```

        return items;
    }

    @SuppressWarnings("unchecked")
    public void addColumns(TableView table) {
        TableColumn<Map, Integer> idCol = new
        TableColumn<>("Id");
        idCol.setCellValueFactory(new
        MapValueFactory<>(idColumnKey));

        TableColumn<Map, String> firstNameCol = new
        TableColumn<>("First Name");
        firstNameCol.setCellValueFactory(new
        MapValueFactory<>(firstNameColumnKey));

        TableColumn<Map, String> lastNameCol = new
        TableColumn<>("Last Name");
        lastNameCol.setCellValueFactory(new
        MapValueFactory<>(lastNameColumnKey));

        TableColumn<Map, LocalDate> birthDateCol = new
        TableColumn<>("Birth Date");
        birthDateCol.setCellValueFactory(new
        MapValueFactory<>(birthDateColumnKey));

        table.getColumns().addAll(idCol, firstNameCol,
        lastNameCol, birthDateCol);
    }
}

```

## Showing and Hiding Columns

By default, all columns in a `TableView` are visible.

The `TableColumn` class has a `visible` property to set the visibility of a column. If you turn off the visibility of a parent column, a column with nested columns, all of its nested columns will also be invisible:

```

TableColumn<Person, String> idCol = new TableColumn<>("Id");

// Make the Id column invisible
idCol.setVisible(false);

...
// Make the Id column visible
idCol.setVisible(true);

```

Sometimes you may want to let the user control the visibility of columns. The `TableView` class has a `tableMenuButtonVisible` property. If it is set to `true`, a menu button is displayed in the header area:

```

// Create a TableView
TableView<Person> table = create the TableView here...

```

```
// Make the table menu button visible
table.setTableMenuButtonVisible(true);
```

Clicking the menu button displays a list of all leaf columns. Columns are displayed as radio menu items that can be used to toggle their visibility. Figure 13-7 shows a `TableView` with four columns.

Its `tableMenuButtonVisible` property is set to true. The figure shows a menu with all column names with a check mark. The menu is displayed when the menu button is clicked. The check marks beside the column names indicate that the columns are visible. Clicking the column name toggles its visibility.

<b>Id</b> ▲▼	<b>First Name</b> ▲▼	<b>Last Name</b> ▼▲	<b>Birth Date</b> ...
2	Advik	Sharan	2012-10-11
1	Ashwin	Sharan	2012-10-11
5	Babalu	Sharan	1980-01-10
3	Layne	Estes	2011-12-16
4	Mason	Boyd	2003-04-20

**Figure 13-7.** A `TableView` with menu button to toggle the visibility of columns

### Reordering Columns in a `TableView`

You can rearrange columns in a `TableView` two ways:

- By dragging and dropping columns to a different position
- By changing their positions in the observable list of returned by the `getColumns()` method of the `TableView` class

The first option is available by default. The user needs to drag and drop a column at the new position. When a column is reordered, its position in the `columns` list is changed. The second option will reorder the column directly in the `columns` list.

There is no easy way to disable the default column-reordering feature. If you want to disable the feature, you would need to add a `ChangeListener` to the `ObservableList` returned by the `getColumns()` method of the `TableView`. When a change is reported, reset the columns so they are in the original order again.

### Sorting Data in a `TableView`

`TableView` has built-in support for sorting data in columns. By default, it allows users to sort data by clicking column headers. It also supports

sorting data programmatically. You can also disable sorting for a column or all columns in a `TableView`.

### Sorting Data by Users

By default, data in all columns in a `TableView` can be sorted. Users can sort data in columns by clicking the column headers. The first click sorts the data in ascending order. The second click sorts the data in descending order. The third click removes the column from the sort order list.

By default, single column sorting is enabled. That is, if you click a column, the records in the `TableView` are sorted based on the data only in the clicked column. To enable multicolumn sorting, you need to press the Shift key while clicking the headers of the columns to be sorted.

`TableView` displays visual clues in the headers of the sorted columns to indicate the sort type and the sort order. By default, a triangle is displayed in the column header indicating the sort type. It points upward for ascending sort type and downward for descending sort type. The sort order of a column is indicated by dots or a number. Dots are used for the first three columns in the sort order list. A number is used for the fourth column onward. For example, the first column in the sort order list displays one dot, the second two dots, the third three dots, the fourth a number 4, the fifth a number 5, and so forth.

Figure 13-8 shows a `TableView` with four columns. The column headers are showing the sort type and sort orders. The sort types are descending for Last Name and ascending for others. The sort orders are 1, 2, 3, and 4 for Last Name, First Name, Birth Date, and Id, respectively. Notice that dots are used for the sort orders in the first three columns and a number 4 is used for the Id column because it is fourth on the sort order list. This sorting is achieved by clicking column headers in the following order: Last Name (twice), First Name, Birth Date, and Id.

<b>Id</b>	<b>First Name</b>	<b>Last Name</b>	<b>Birth Date</b>	+
1	Ashwin	Sharan	2012-10-11	✓ Id
2	Advik	Sharan	2012-10-11	✓ First Name
3	Layne	Estes	2011-12-16	✓ Last Name
4	Mason	Boyd	2003-04-20	✓ Birth Date
5	Babalu	Sharan	1980-01-10	

Figure 13-8. Column headers showing the sort type and sort order

## Sorting Data Programmatically

Data in columns can be sorted programmatically.

The `TableView` and  `TableColumn` classes provide a very powerful API for sorting. The sorting API consists of several properties and methods in the two classes. Every part and every stage of sorting are customizable. The following sections describe the API with examples.

### Making a Column Sortable

The `sortable` property of a  `TableColumn` determines whether the column is sortable. By default, it is set to true. Set it to false to disable the sorting for a column:

```
// Disable sorting for fNameCol column  
fNameCol.setSortable(false);
```

### Specifying the Sort Type of a Column

A  `TableColumn` has a sort type, which can be ascending or descending. It is specified through the `sortType` property.

The `ASCENDING` and `DESCENDING` constants

of  `TableColumn.SortType` enum represent the ascending and descending, respectively, sort types for columns. The default value for the `sortType` property is  `TableColumn.SortType.ASCENDING`.

The `DESCENDING` constant is set as follows:

```
// Set the sort type for fNameCol column to descending  
fNameCol.setSortType(TableColumn.SortType.DESCENDING);
```

### Specifying the Comparator for a Column

A  `TableColumn` uses a  `Comparator` to sort its data. You can specify the  `Comparator` for a  `TableColumn` using its  `comparator` property. The  `comparator` is passed in the objects in two cells being compared. A  `TableColumn` uses a default  `Comparator`, which is represented by the constant  `TableColumn.DEFAULT_COMPARATOR`. The default  `comparator` compares data in two cells using the following rules:

- It checks for  `null` values. The  `null` values are sorted first. If both cells have  `null`, they are considered equal.
- If the first value being compared is an instance of the  `Comparable` interface, it calls the  `compareTo()` method of the first object passing the second object as an argument to the method.

- If neither of the above two conditions are true, it converts the two objects into strings calling their `toString()` methods and uses a `Collator` to compare the two `String` values.

In most cases, the default comparator is sufficient. The following snippet of code uses a custom comparator for a `String` column that compares only the first characters of the cell data:

```
TableColumn<Person, String> fNameCol = new TableColumn<>("First Name");
...
// Set a custom comparator
fNameCol.setComparator((String n1, String n2) -> {
    if (n1 == null && n2 == null) {
        return 0;
    }

    if (n1 == null) {
        return -1;
    }

    if (n2 == null) {
        return 1;
    }

    String c1 = n1.isEmpty() ? n1:String.valueOf(n1.charAt(0));
    String c2 = n2.isEmpty() ? n2:String.valueOf(n2.charAt(0));
    return c1.compareTo(c2);
});
```

## Specifying the Sort Node for a Column

The `TableColumn` class contains a `sortNode` property, which specifies a node to display a visual clue in the column header about the current sort type and sort order for the column. The node is rotated by 180 degrees when the sort type is ascending. The node is invisible when the column is not part of the sort. By default, it is `null` and the `TableColumn` provides a triangle as the sort node.

## Specifying the Sort Order of Columns

The `TableView` class contains several properties that are used in sorting. To sort columns, you need to add them to sort order list of the `TableView`. The `sortOrder` property specifies the sort order. It is an `ObservableList` of `TableColumn`. The order of a `TableColumn` in the list specifies the order of the column in the sort. Rows are sorted based on the first column in the list. If values in two rows in the column are equal, the second column in the sort order list is used to determine the sort order of the two rows and so on.

The following snippet of code adds two columns to a `TableView` and specifies their sort order. Notice that both columns will be sorted in ascending order, which is the default sort type. If you want to sort them in descending order, set their `sortType` property as follows:

```
// Create a TableView with data
TableView<Person> table = new
TableView<>(PersonTableUtil.getPersonList());

TableColumn<Person, String> lNameCol
= PersonTableUtil.getLastColumnName();
TableColumn<Person, String> fNameCol
= PersonTableUtil.getFirstNameColumn();

// Add columns to the TableView
table.getColumns().addAll(lNameCol, fNameCol);

// Add columns to the sort order to sort by last name followed by
first name
table.getSortOrder().addAll(lNameCol, fNameCol);
```

The `sortOrder` property of the `TableView` is monitored for changes. If it is modified, the `TableView` is sorted immediately based on the new sort order. Adding a column to a sort order list does not guarantee inclusion of the column in sorting. The column must also be sortable to be included in sorting. The `sortType` property of the `TableColumn` is also monitored for changes. Changing the sort type of a column, which are in the sort order list, resorts the `TableView` data immediately.

## Getting the Comparator for a `TableView`

`TableView` contains a read-only `comparator` property, which is a `Comparator` based on the current sort order list. You rarely need to use this `Comparator` in your code. If you pass two `TableView` items to the `compare()` method of the `Comparator`, it will return a negative integer, zero, or a positive integer indicating that the first item is less than, equal to, or greater than the second item, respectively.

Recall that  `TableColumn` also has a `comparator` property, which is used to specify how to determine the order of values in the cells of the `TableColumn`. The `comparator` property of the `TableView` combines the `comparator` properties of all `TableColumns` in its sort order list.

## Specifying the Sort Policy

A `TableView` has a sort policy to specify how the sorting is performed. It is a `Callback` object. The `TableView` is passed in as an argument to

the `call()` method. The method returns `true` if the sorting successes. It returns `false` or `null` if the sorting fails.

The `TableView` class contains a `DEFAULT_SORT_POLICY` constant, which is used as a default sort policy for a `TableView`. It sorts the items list of the `TableView` using its `comparator` property. Specify a sort policy to take full charge of the sorting algorithm. The `call()` method of the sort policy `Callback` object will perform the sorting of the items of the `TableView`.

As a trivial example, setting the sort policy to `null` will disable the sorting, as no sorting will be performed when sorting is requested by the user or program:

```
TableView<Person> table = ...

// Disable sorting for the TableView
table.setSortPolicy(null);
```

Sometimes it is useful to disable sorting temporarily for performance reasons. Suppose you have a sorted `TableView` with a large number of items and you want to make several changes to the sort order list. Every change in the sort order list will trigger a sort on the items. In this case, you may disable the sorting by setting the sort policy to `null`, make all your changes, and enable the sorting by restoring the original sort policy. A change in the sort policy triggers an immediate sort. This technique will sort the items only once:

```
TableView<Person> table = ...

...
// Store the current sort policy
Callback<TableView<Person>, Boolean> currentSortPolicy
= table.getSortPolicy();

// Disable the sorting
table.setSortPolicy(null)

// Make all changes that might need or trigger sorting
...

// Restore the sort policy that will sort the data once
// immediately
table.setSortPolicy(currentSortPolicy);
```

## Sorting Data Manually

`TableView` contains a `sort()` method that sorts the items in the `TableView` using the current sort order list. You may call this method to sort items after adding a number of items to a `TableView`. This method is automatically called when the sort type of a column, the sort order, or sort policy changes.

## Handling Sorting Event

`TableView` fires a `SortEvent` when it receives a request for sorting and just before it applies the sorting algorithm to its items. Add a `SortEvent` listener to perform any action before the actual sorting is performed:

```
TableView<Person> table = ...
table.setOnSort(e -> {/* Code to handle the sort event */});
```

If the `SortEvent` is consumed, the sorting is aborted. If you want to disable sorting for a `TableView`, consume the `SortEvent` as follows:

```
// Disable sorting for the TableView
table.setOnSort(e -> e.consume());
```

## Disabling Sorting for a TableView

There are several ways you can disable sorting for a `TableView`.

- Setting the `sortable` property for a `TableColumn` disables sorting only for that column. If you set the `sortable` property to false for all columns in a `TableView`, the sorting for the `TableView` is disabled.
- You can set the sort policy for the `TableView` to null.
- You can consume the `SortEvent` for the `TableView`.
- Technically, it is possible, though not recommended, to override the `sort()` method of the `TableView` class and provide an empty body for the method.

The best way to disable sorting partially or completely for a `TableView` is to disable sorting for some or all of its columns.

## Customizing Data Rendering in Cells

A cell in a  `TableColumn` is an instance of the  `TableCell` class, which displays the data in the cell. A  `TableCell` is a Labeled control, which is capable of displaying text, a graphic, or both.

You can specify a cell factory for a  `TableColumn`. The job of a cell factory is to render the data in the cell. The  `TableColumn` class contains a `cellFactory` property, which is a  `Callback` object. Its `call()` method is passed in the reference of the  `TableColumn` to which the cell belongs. The method returns an instance of  `TableCell`. The `updateItem()` method of the  `TableCell` is overridden to provide the custom rendering of the cell data.

`TableColumn` uses a default cell factory if its `cellFactory` property is not specified. The default cell factory

displays the cell data depending on the type of the data. If the cell data comprise a node, the data are displayed in the `graphic` property of the cell. Otherwise, the `toString()` method of the cell data is called and the returned string is displayed in the `text` property of the cell.

Up to this point, you have been using a list of `Person` objects as the data model in the examples for displaying data in a `TableView`. The Birth Date column is formatted as `yyyy-mm-dd`, which is the default ISO date format return by the `toString()` method of the `LocalDate` class. If you would like to format birth dates in the `mm/dd/yyyy` format, you can achieve this by setting a custom cell factory for the Birth Date column:

```
TableColumn<Person, LocalDate> birthDateCol = ...;
birthDateCol.setCellFactory (col -> {
    TableCell<Person, LocalDate> cell = new TableCell<Person,
LocalDate> () {
        @Override
        public void updateItem(LocalDate item, boolean
empty) {
            super.updateItem(item, empty);

            // Cleanup the cell before populating it
            this.setText(null);
            this.setGraphic(null);

            if (!empty) {
                // Format the birth date in mm/dd/yyyy
format
                String formattedDob =
                    DateTimeFormatter.ofPattern("MM/dd
/yyyy").format(item);
                this.setText(formattedDob);
            }
        }
    };
    return cell;
});
```

You can also use the above technique to display images in cells. In the `updateItem()` method, create an `ImageView` object for the image and display it using the `setGraphic()` method of the `TableCell`. `TableCell` contains `tableColumn`, `tableRow`, and `tableView` properties that store the references of its  `TableColumn`,  `TableRow`, and  `TableView`, respectively. These properties are useful to access the item in the data model that represents the row for the cell.

If you replace the `if` statement in the above snippet of code with the following code, the Birth Date column displays the birth date and age category, for example, `10/11/2012 (BABY)`:

```

if (!empty) {
    String formattedDob
= DateTimeFormatter.ofPattern("MM/dd/yyyy").format(item);

    if (this.getTableRow() != null) {
        // Get the Person item for this cell
        int rowIndex = this.getTableRow().getIndex();
        Person p =
this.getTableview().getItems().get(rowIndex);
        String ageCategory = p.getAgeCategory().toString();

        // Display birth date and age category together
        this.setText(formattedDob + " (" + ageCategory + ")"
    );
}
}

```

The following are subclasses of `TableCell` that render cell data in different ways. For example, a `CheckBoxTableCell` renders cell data in a check box and a `ProgressBarTableCell` renders a number using a progress bar:

- `CheckBoxTableCell`
- `ChoiceBoxTableCell`
- `ComboBoxTableCell`
- `ProgressBarTableCell`
- `TextFieldTableCell`

The following snippet of code creates a column labeled Baby? and sets a cell factory to display the value in a `CheckBoxTableCell`.

The `forTableColumn( TableColumn<S, Boolean> col)` method of the `CheckBoxTableCell` class returns a `Callback` object that is used as a cell factory:

```

// Create a "Baby?" column
TableColumn<Person, Boolean> babyCol = new
TableColumn<>("Baby?");
babyCol.setCellValueFactory(cellData -> {
    Person p = cellData.getValue();
    Boolean v = (p.getAgeCategory() ==
Person.AgeCategory.BABY);
    return new ReadOnlyBooleanWrapper(v);
});

// Set a cell factory that will use a CheckBox to render the
value
babyCol.setCellFactory(CheckBoxTableCell.<Person>forTableColumn(b
abyCol));

```

Please explore the API documentation for other subclasses of the `TableCell` and how to use them. For example, you can display a

combo box with a list of choices in the cells of a column. Users can select one of the choices as the cell data.

**Listing 13-5** has a complete program to show how to use custom cell factories. It displays a window as shown in **Figure 13-9**. The program uses a cell factory to format the birth date in mm/dd/yyyy format and a cell factory to display whether a person is a baby using a check box.

### ***Listing 13-5.*** Using a Custom Cell Factory for a TableColumn

```
// TableViewCellFactoryTest.java
package com.jdojo.control;

import com.jdojo.mvc.model.Person;
import javafx.application.Application;
import java.time.LocalDate;
import javafx.scene.Scene;
import javafx.scene.control.TableCell;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;
import java.time.format.DateTimeFormatter;
import javafx.beans.property.ReadOnlyBooleanWrapper;
import javafx.scene.control.cell.CheckBoxTableCell;

public class TableViewCellFactoryTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    @SuppressWarnings("unchecked")
    public void start(Stage stage) {
        TableView<Person> table = new
        TableView<>(PersonTableUtil.getPersonList());

        // Create the birth date column
        TableColumn<Person, LocalDate> birthDateCol =
            PersonTableUtil.getBirthDateColumn();

        // Set a custom cell factory for Birth Date column
        birthDateCol.setCellFactory(col -> {
            TableCell<Person, LocalDate> cell = new
            TableCell<Person, LocalDate>() {
                @Override
                public void updateItem(LocalDate item,
                boolean empty) {
                    super.updateItem(item, empty);

                    // Cleanup the cell before
                    this.setText(null);
                }
            };
        });
    }
}
```

```

        this.setGraphic(null);

        if (!empty) {
            String formattedDob =
                DateTimeFormatter.ofPattern
                    ("MM/dd/yyyy")
                        .format(item);
            this.setText(formattedDob);
        }
    }
}

return cell;
} );

// Create and configure the baby column
TableColumn<Person, Boolean> babyCol = new
TableColumn<>("Baby?");
babyCol.setCellValueFactory(
    cellData -> {
        Person p = cellData.getValue();
        Boolean v =
            (p.getAgeCategory() ==
        Person.AgeCategory.BABY);
        return new
ReadOnlyBooleanWrapper(v);
    });

// Set a custom cell factory for the baby column
babyCol.setCellFactory(
    CheckBoxTableCell.<Person>forTableColumn
(babyCol));

// Add columns to the table
table.getColumns().addAll(PersonTableUtil.getIdColum
n(),
    PersonTableUtil.getFirstNameColumn(),
    PersonTableUtil.getLastNameColumn(),
    birthDateCol,
    babyCol);

HBox root = new HBox(table);
root.setStyle("-fx-padding: 10;" +
    "-fx-border-style: solid inside;" +
    "-fx-border-width: 2;" +
    "-fx-border-insets: 5;" +
    "-fx-border-radius: 5;" +
    "-fx-border-color: blue;");

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("Using a Custom Cell Factory for
a TableColumn");
stage.show();

```

```

    }
}

<!-- Using a Custom Cell Factory for a TableColumn -->
<TableColumn id="Baby" title="Baby?" type="Boolean" />

```

The screenshot shows a JavaFX application window titled "Using a Custom Cell Factory for a TableColumn...". Inside the window, there is a TableView with a blue border. The table has five rows and five columns. The columns are labeled "Id", "First Name", "Last Name", "Birth Date", and "Baby?". The "Baby?" column contains five empty check boxes. The data in the table is as follows:

<b>Id</b>	<b>First Name</b>	<b>Last Name</b>	<b>Birth Date</b>	<b>Baby?</b>
1	Ashwin	Sharan	10/11/2012	<input type="checkbox"/>
2	Advik	Sharan	10/11/2012	<input type="checkbox"/>
3	Layne	Estes	12/16/2011	<input type="checkbox"/>
4	Mason	Boyd	04/20/2003	<input type="checkbox"/>
5	Babalu	Sharan	01/10/1980	<input type="checkbox"/>

**Figure 13-9.** Using custom cell factories to format data in cells and display cell data in check boxes

## Selecting Cells and Rows in a *TableView*

*TableView* has a selection model represented by its property `selectionModel`. A selection model is an instance of the `TableViewSelectionModel` class, which is an inner static class of the `TableView` class. The selection model supports cell-level and row-level selection. It also supports two selection modes: single and multiple. In the single-selection mode, only one cell or row can be selected at a time. In the multiple-selection mode, multiple cells or rows can be selected. By default, single-row selection is enabled. You can enable multirow selection, as follows:

```

TableView<Person> table = ...

// Turn on multiple-selection mode for the TableView
TableViewSelectionModel<Person> tsm = table.getSelectionModel();
tsm.setSelectionMode(SelectionMode.MULTIPLE);

```

The cell-level selection can be enabled by setting the `cellSelectionEnabled` property of the selection model to true, as in the following snippet of code. When the property is set to true, the `TableView` is put in cell-level selection mode and you cannot select an entire row. If multiple-selection mode is enabled, you can still select all cells in a row. However, the row itself is not reported as selected as

the TableView is in the cell-level selection mode. By default, cell-level selection mode is false.

```
// Enable cell-level selection
tsm.setCellSelectionEnabled(true);
```

The selection model provides information about the selected cells and rows. The `isSelected(int rowIndex)` method returns `true` if the row at the specified `rowIndex` is selected. Use the `isSelected(int rowIndex, TableColumn<S, ?> column)` method to know if a cell at the specified `rowIndex` and `column` is selected. The selection model provides several methods to select cells and rows and get the report of selected cells and rows:

- The `selectAll()` method selects all cells or rows.
- The `select()` method is overloaded. It selects a row, a row for an item, and a cell.
- The `isEmpty()` method returns `true` if there is no selection. Otherwise, it returns `false`.
- The `getSelectedCells()` method returns a read-only `ObservableList<TablePosition>` that is the list of currently selected cells. The list changes as the selection in the TableView changes.
- The `getSelectedIndices()` method returns a read-only `ObservableList<Integer>` that is the list of currently selected indices. The list changes as the selection in the TableView changes. If row-level selection is enabled, an item in the list is the row index of the selected row. If cell-level selection is enabled, an item in the list is the row index of the row in which one or more cells are selected.
- The `getSelectedItems()` method returns a read-only `ObservableList<S>` where `S` is the generic type of the TableView. The list contains all items for which the corresponding row or cells have been selected.
- The `clearAndSelect()` method is overloaded. It lets you clear all selections before selecting a row or a cell.
- The `clearSelection()` method is overloaded. It lets you clear selections for a row, a cell, or the entire TableView.

It is often a requirement to make some changes or take an action when a cell or row selection changes in a TableView. For example, a TableView may act as a master list in a master-detail data view. When the user selects a row in the master list, you want to refresh the data in the detail view. If you are interested in handling the selection change

event, you need to add a `ListChangeListener` to one of the `ObservableLists` returned by the above listed methods that reports on the selected cells or rows. The following snippet of code adds a `ListChangeListener` to the `ObservableList` returned by the `getSelectedIndices()` method to track the row selection change in a `TableView`:

```
TableView<Person> table = ...
TableViewSelectionModel<Person> tsm = table.getSelectionModel();
ObservableList<Integer> list = tsm.getSelectedIndices();

// Add a ListChangeListener
list.addListener((ListChangeListener.Change<? extends Integer>
change) -> {
    System.out.println("Row selection has changed");
});
```

## Editing Data in a `TableView`

A cell in a `TableView` can be edited. An editable cell switches between editing and nonediting modes. In editing mode, cell data can be modified by the user. For a cell to enter editing mode, the `TableView`,  `TableColumn`, and  `TableCell` must be editable. All three of them have an `editable` property, which can be set to true using the `setEditable(true)` method. By default,  `TableColumn` and  `TableCell` are editable. To make cells editable in a `TableView`, you need make the `TableView editable`:

```
TableView<Person> table = ...
table.setEditable(true);
```

The  `TableColumn` class supports three types of events:

- `onEditStart`
- `onEditCommit`
- `onEditCancel`

The `onStartEdit` event is fired when a cell in the column enters editing mode. The `onEditCommit` event is fired when the user successfully commits the editing, for example, by pressing the Enter key in a `TextField`. The `onEditCancel` event is fired when the user cancels the editing, for example, by pressing the Esc key in a `TextField`.

The events are represented by an object of the  `TableColumn.CellEditEvent` class. The event object encapsulates the old and new values in the cell, the row object from the items list of the  `TableView`,  `TableColumn`,  `TablePosition` indicating the cell

position where the editing is happening, and the reference of the `TableView`. Use the methods of the `CellEditEvent` class to get these values.

Making a `TableView` editable does not let you edit its cell data. You need to do a little more plumbing before you can edit data in cells. Cell-editing capability is provided through specialized implementation of the `TableCell` class. the JavaFX library provides a few of these implementations. Set the cell factory for a column to use one of the following implementations of the `TableCell` to edit cell data:

- `CheckBoxTableCell`
- `ChoiceBoxTableCell`
- `ComboBoxTableCell`
- `TextFieldTableCell`

### Editing Data Using a Check Box

A `CheckBoxTableCell` renders a check box inside the cell. Typically it is used to represent a boolean value in a column. The class provides a way to map other types of values to a boolean value using a `Callback` object. The check box is selected if the value is true. Otherwise, it is unselected. Bidirectional binding is used to bind the selected property of the check box and the underlying `ObservableValue`. If the user changes the selection, the underlying data are updated and vice versa.

You do not have a boolean property in the `Person` class. You must create a boolean column by providing a cell value factory, as shown in the following code. If a `Person` is a baby, the cell value factory returns `true`. Otherwise, it returns `false`.

```
TableColumn<Person, Boolean> babyCol = new
TableColumn<>("Baby?");
babyCol.setCellValueFactory(cellData -> {
    Person p = cellData.getValue();
    Boolean v = (p.getAgeCategory() ==
Person.AgeCategory.BABY);
    return new ReadOnlyBooleanWrapper(v);
});
```

Getting a cell factory to use `CheckBoxTableCell` is easy. Use the `forTableColumn()` static method to get a cell factory for the column:

```
// Set a CheckBoxTableCell to display the value
babyCol.setCellFactory(CheckBoxTableCell.<Person>forTableColumn(b
abyCol));
```

A CheckBoxTableCell does not fire the cell-editing events. The selected property of the check box is bound to the ObservableValue representing the data in the cell. If you are interested in tracking the selection change event, you need to add a ChangeListener to the data for the cell.

## Editing Data Using a Choice Box

A ChoiceBoxTableCell renders a choice box with a specified list of values inside the cell. The type of values in the list must match the type of the TableColumn. The data in a ChoiceBoxTableCell are displayed in a Label when the cell is not being edited. A ChoiceBox is used when the cell is being edited.

The Person class does not have a gender property. You want to add a Gender column to a TableView<Person>, which can be edited using a choice box. The following snippet of code creates the TableColumn and sets a cell value factory, which sets all cells to an empty string. You would set the cell value factory to use the gender property of the Person class if you had one.

```
// Gender is a String, editable, ComboBox column
TableColumn<Person, String> genderCol = new
TableColumn<>("Gender");

// Use an appropriate cell value factory.
// For now, set all cells to an empty string
genderCol.setCellValueFactory(cellData -> new
ReadOnlyStringWrapper(""));
```

You can create a cell factory that uses a choice box for editing data in cells using the forTableColumn() static method of the ChoiceBoxTableCell class. You need to specify the list of items to be displayed in the choice box.

```
// Set a cell factory, so it can be edited using a ChoiceBox
genderCol.setCellFactory(
    ChoiceBoxBoxTableCell.<Person,
    String>forTableColumn("Male", "Female"));
```

When an item is selected in the choice box, the item is set to the underlying data model. For example, if a column is based on a property in the domain object, the selected item will be set to the property. You can set an onEditCommit event handler that is fired when the user selects an item. The following snippet of code adds such a handler for the Gender column that prints a message on the standard output:

```
// Add an onEditCommit handler
genderCol.setOnEditCommit(e -> {
    int row = e.getTablePosition().getRow();
    Person person = e.getRowValue();
    System.out.println("Gender changed ("
```

```
+ person.getFirstName() + " " +
           person.getLastName() + ")" + " at row "
+ (row + 1) +
    ". New value = " + e.getNewValue());
});
```

Clicking a selected cell puts the cell into editing mode. Double-clicking an unselected cell puts the cell into editing mode. Changing the focus to another cell or selecting an item from the list puts the editing cell into nonediting mode and the current value is displayed in a Label.

## Editing Data Using a Combo Box

A ComboBoxTableCell renders a combo box with a specified list of values inside the cells. It works similar to a ChoiceBoxTableCell. Please refer to the section “Editing Data Using a Choice Box” for more details.

## Editing Data Using a *TextField*

A TextFieldTableCell renders a TextField inside the cell when the cell is being edited where the user can modify the data. It renders the cell data in a Label when the cell is not being edited.

Clicking a selected cell or double-clicking an unselected cell puts the cell into editing mode, which displays the cell data in a TextField. Once the cell is in editing mode, you need to click in the TextField (one more click!) to put the caret in the TextField so you can make changes. Notice that you need a minimum of three clicks to edit a cell, which is a pain for those users who have to edit a lot of data. Let’s hope that the designers of the TableView API will make data editing less cumbersome in future releases.

If you are in the middle of editing a cell data, press the Esc key to cancel editing, which will return the cell to nonediting mode and reverts to the old data in the cell. Pressing the Enter key commits the data to the underlying data model if the TableColumn is based on a WritableObservableValue.

If you are editing a cell using a TextFieldTableCell, moving the focus to another cell, for example, by clicking another cell, cancels the editing and puts the old value back in the cell. This is not what a user expects. At present, there is no easy solution for this problem. You will have to create a subclass of TableCell and add a focus change listener, so you can commit the data when the TextField loses focus.

Use the `forTableColumn()` static method of the `TextFieldTableCell` class to get a cell factory that uses

a `TextField` to edit cell data. The following snippet of code shows how to do it for a First Name String column:

```
 TableColumn<Person, String> fNameCol = new TableColumn<>("First Name");
fNameCol.setCellFactory(TextFieldTableCell.<Person>forTableColumn());
```

Sometimes you need to edit nonstring data using a `TextField`, for example, for a date. The date may be represented as an object of the `LocalDate` class in the model. You may want to display it in a `TextField` as a formatted string. When the user edits the date, you want to commit the data to the model as a `LocalDate`.

The `TextFieldTableCell` class supports this kind of object-to-string and vice versa conversion through a `StringConverter`. The following snippet of code sets a cell factory for a Birth Date column with a `StringConverter`, which converts a string to a `LocalDate` and vice versa. The column type is `LocalDate`. By default, the `LocalDateStringConverter` assumes a date format of `mm/dd/yyyy`.

```
 TableColumn<Person, LocalDate> birthDateCol = new TableColumn<>("Birth Date");
LocalDateStringConverter converter = new LocalDateStringConverter();
birthDateCol.setCellFactory(TextFieldTableCell.<Person, LocalDate>forTableColumn(converter));
```

The program in Listing 13-6 shows how to edit data in a `TableView` using different types of controls. The `TableView` contains Id, First Name, Last Name, Birth Date, Baby, and Gender columns. The Id column is noneditable. The First Name, Last Name, and Birth Date columns use `TextFieldTableCell`, so they can be edited using a `TextField`. The Baby column is a noneditable computed field and is not backed by the data model. It uses `CheckBoxTableCell` to render its values. The Gender column is an editable computed field. It is not backed by the data model. It uses a `ComboBoxTableCell` that presents the user a list of values (Male and Female) in editing model. When the user selects a value, the value is not saved to the data model. It stays in the cell. An `onEditCommit` event handler is added that prints the gender selection on the standard output. The program displays a window as shown in Figure 13-10, where it can be seen that you have already selected a gender value for all persons. The Birth Date value for the fifth row is being edited.

### ***Listing 13-6.*** Editing Data in a `TableView`

```
// TableViewEditing.java
package com.jdojo.control;

import com.jdojo.mvc.model.Person;
import javafx.application.Application;
import javafx.beans.property.ReadOnlyBooleanWrapper;
import javafx.scene.Scene;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.cell.TextFieldTableCell;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;
import java.time.LocalDate;
import javafx.beans.property.ReadOnlyStringWrapper;
import javafx.scene.control.cell.CheckBoxTableCell;
import javafx.scene.control.cell.ComboBoxTableCell;

public class TableViewEditing extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        TableView<Person> table = new
TableView<>(PersonTableUtil.getPersonList());

        // Make the TableView editable
        table.setEditable(true);

        // Add columns with appropriate editing features
        addIdColumn(table);
        addFirstNameColumn(table);
        addLastNameColumn(table);
        addBirthDateColumn(table);
        addBabyColumn(table);
        addGenderColumn(table);

        HBox root = new HBox(table);
        root.setStyle("-fx-padding: 10;" +
                     "-fx-border-style: solid inside;" +
                     "-fx-border-width: 2;" +
                     "-fx-border-insets: 5;" +
                     "-fx-border-radius: 5;" +
                     "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Editing Data in a TableView");
        stage.show();
    }

    public void addIdColumn(TableView<Person> table) {
        // Id column is non-editable
        table.getColumns().add(PersonTableUtil.getIdColumn()
```

```

) ;

}

public void addFirstNameColumn(TableView<Person> table) {
    // First Name is a String, editable column
    TableColumn<Person, String> fNameCol
= PersonTableUtil.getFirstNameColumn();

    // Use a TextFieldTableCell, so it can be edited
    fNameCol.setCellFactory(TextFieldTableCell.<Person>forTableColumn());

    table.getColumns().add(fNameCol);
}

public void addLastNameColumn(TableView<Person> table) {
    // Last Name is a String, editable column
    TableColumn<Person, String> lNameCol
= PersonTableUtil.getLastNameColumn();

    // Use a TextFieldTableCell, so it can be edited
    lNameCol.setCellFactory(TextFieldTableCell.<Person>forTableColumn());

    table.getColumns().add(lNameCol);
}

public void addBirthDateColumn(TableView<Person> table) {
    // Birth Date is a LocalDate, editable column
    TableColumn<Person, LocalDate> birthDateCol =
        PersonTableUtil.getBirthDateColumn();

    // Use a TextFieldTableCell, so it can be edited
    LocalDateStringConverter converter = new
LocalDateStringConverter();
    birthDateCol.setCellFactory(
        TextFieldTableCell.<Person,
LocalDate>forTableColumn(converter));

    table.getColumns().add(birthDateCol);
}

public void addBabyColumn(TableView<Person> table) {
    // Baby? is a Boolean, non-editable column
    TableColumn<Person, Boolean> babyCol = new
TableColumn<>("Baby?");
    babyCol.setEditable(false);

    // Set a cell value factory
    babyCol.setCellValueFactory(cellData -> {
        Person p = cellData.getValue();
        Boolean v = (p.getAgeCategory() ==
Person.AgeCategory.BABY);
        return new ReadOnlyBooleanWrapper(v);
    });
}

```

```
// Use a CheckBoxTableCell to display the boolean
value
    babyCol.setCellFactory(
        CheckBoxTableCell.<Person>forTableColumn(babyC
ol));
    table.getColumns().add(babyCol);
}

public void addGenderColumn(TableView<Person> table) {
    // Gender is a String, editable, ComboBox column
    TableColumn<Person, String> genderCol = new
    TableColumn<>("Gender");
    genderCol.setMinWidth(80);

    // By default, all cells are have null values
    genderCol.setCellValueFactory(
        cellData -> new ReadOnlyStringWrapper(null));

    // Set a ComboBoxTableCell, so you can selects
    // a value from a list
    genderCol.setCellFactory(
        ComboBoxTableCell.<Person,
        String>forTableColumn("Male", "Female"));

    // Add an event handler to handle the edit commit
    // event.
    // It displays the selected value on the standard
    // output
    genderCol.setOnEditCommit(e -> {
        int row = e.getTablePosition().getRow();
        Person person = e.getRowValue();
        System.out.println("Gender changed for " +
            person.getFirstName() + " "
+ person.getLastName() +
            " at row " + (row + 1) + " to "
+ e.getNewValue());
    });
    table.getColumns().add(genderCol);
}
}
```

<b>Id</b>	<b>First Name</b>	<b>Last Name</b>	<b>Birth Date</b>	<b>Baby?</b>	<b>Gender</b>
1	Ashwin	Sharan	10/11/2012	<input type="checkbox"/>	Male
2	Advik	Sharan	10/11/2012	<input type="checkbox"/>	Male
3	Layne	Estes	12/16/2011	<input type="checkbox"/>	Female
4	Mason	Boyd	04/20/2003	<input type="checkbox"/>	Male
5	Babalu	Sharan	01/10/1980	<input type="checkbox"/>	Male

**Figure 13-10.** A TableView with a cell in editing mode

### Editing Data in TableCell Using any Control

In the previous section, I discussed editing data in cells of a TableView using different controls, for example, TextField, CheckBox, and ChoiceBox. You can subclass TableCell to use any control to edit cell data. For example, you may want to use a DatePicker to select a date in cells of a date column or RadioButtons to select from multiple options. The possibilities are endless.

You need to override four methods of the TableCell class:

- startEdit()
- commitEdit()
- cancelEdit()
- updateItem()

The startEdit() method for the cell transitions from nonediting mode to editing mode. Typically, you set the control of your choice in the graphic property of the cell with the current data.

The commitEdit() method is called when the user action, for example, pressing the Enter key in a TextField, indicates that the user is done modifying the cell data and the data need to be saved in the underlying data model. Typically, you do not need to override this method as the modified data are committed to the data model if the TableColumn is based on a Writable ObservableValue.

The `cancelEdit()` method is called when the user action, for example, pressing the Esc key in a `TextField`, indicates that the user wants to cancel the editing process. When the editing process is canceled, the cell returns to nonediting mode. You need to override this method and revert the cell data to their old values.

The `updateItem()` method is called when the cell needs to be rendered again. Depending on the editing mode, you need to set the text and graphic properties of the cell appropriately.

Now let's develop a `DatePickerTableCell` class that inherits from the `TableCell` class. You can use instances of `DatePickerTableCell` when you want to edit cells of a `TableColumn` using a `DatePicker` control. The `TableColumn` must be of `LocalDate`. Listing 13-7 has the complete code for the `DatePickerTableCell` class.

### ***Listing 13-7. The DatePickerTableCell Class to Allows Editing Table Cells Using a DatePickerControl***

```
// DatePickerTableCell.java
package com.jdojo.control;

import javafx.beans.value.ObservableValue;
import javafx.scene.control.DatePicker;
import javafx.scene.control.TableCell;
import javafx.beans.value.ChangeListener;
import javafx.scene.control.TableColumn;
import javafx.util.Callback;
import javafx.util.StringConverter;

@SuppressWarnings("unchecked")
public class DatePickerTableCell<S, T> extends TableCell<S, java.time.LocalDate> {
    private DatePicker datePicker;
    private StringConverter converter = null;
    private boolean datePickerEditable = true;

    public DatePickerTableCell() {
        this.converter = new LocalDateStringConverter();
    }

    public DatePickerTableCell(boolean datePickerEditable) {
        this.converter = new LocalDateStringConverter();
        this.datePickerEditable = datePickerEditable;
    }

    public DatePickerTableCell(StringConverter<java.time.LocalDate> converter) {
        this.converter = converter;
```

```

    }

    public
DatePickerTableCell(StringConverter<java.time.LocalDate>
converter,
                     boolean datePickerEditable) {
    this.converter = converter;
    this.datePickerEditable = datePickerEditable;
}

@Override
public void startEdit() {
    // Make sure the cell is editable
    if (!isEditable() ||
        !getTableView().isEditable() ||
        !get TableColumn().isEditable()) {
        return;
    }

    // Let the ancestor do the plumbing job
    super.startEdit();

    // Create a DatePicker, if needed, and set it as the
graphic for the cell
    if (datePicker == null) {
        this.createDatePicker();
    }

    this.setGraphic(datePicker);
}

@Override
public void cancelEdit() {
    super.cancelEdit();
    this.setText(converter.toString(this.getItem()));
    this.setGraphic(null);
}

@Override
public void updateItem(java.time.LocalDate item, boolean
empty) {
    super.updateItem(item, empty);

    // Take actions based on whether the cell is being
edited or not
    if (empty) {
        this.setText(null);
        this.setGraphic(null);
    } else {
        if (this.isEditing()) {
            if (datePicker != null) {
                datePicker.setValue((java.time.LocalDate)item);
            }
        }
        this.setText(null);
    }
}

```

```

        this.setGraphic(datePicker);
    } else {
        this.setText(converter.toString(item));
        this.setGraphic(null);
    }
}

private void createDatePicker() {
    datePicker = new DatePicker();
    datePicker.setConverter(converter);

    // Set the current value in the cell to the
    DatePicker
    datePicker.setValue((java.time.LocalDate)this.getIte
m());

    // Configure the DatePicker properties
    datePicker.setPrefWidth(this.getWidth() -
this.getGraphicTextGap() * 2);
    datePicker.setEditable(this.datePickerEditable);

    // Commit the new value when the user selects or
enters a date
    datePicker.valueProperty().addListener(new
ChangeListener() {
        @Override
        public void changed(ObservableValue prop,
                            Object oldValue,
                            Object newValue) {
            if
(DatePickerTableCell.this.isEditing()) {
                DatePickerTableCell.this.commitEdi
t(
                    (java.time.LocalDate)newValue);
            }
        }
    });
}

public static <S> Callback<TableColumn<S,
java.time.LocalDate>,
                    TableCell<S, java.time.LocalDate>>
for TableColumn() {
    return for TableColumn(true);
}

public static <S> Callback<TableColumn<S,
java.time.LocalDate>,
                    TableCell<S, java.time.LocalDate>>
for TableColumn(boolean datePickerEditable) {
    return (col -> new
DatePickerTableCell<>(datePickerEditable));
}

```

```

        public static <S> Callback<TableColumn<S,
java.time.LocalDate>, TableCell<S, java.time.LocalDate>>
forTableColumn(StringConverter<java.time.LocalDate> converter) {
    return forTableColumn(converter, true);
}

        public static <S> Callback<TableColumn<S,
java.time.LocalDate>, TableCell<S, java.time.LocalDate>>
forTableColumn(StringConverter<java.time.LocalDate> converter,
boolean datePickerEditable) {
    return (col -> new DatePickerTableCell<>(converter,
datePickerEditable));
}
}

```

The `DatePickerTableCell` class supports a `StringConverter` and the `editable` property value for the `DatePicker`. You can pass them to the constructors or the `forTableColumn()` methods. It creates a `DatePicker` control when the `startEdit()` method is called for the first time. A `ChangeListener` is added that commits the data when a new date is entered or selected. Several versions of the `forTableColumn()` static methods are provided that return cell factories. The following snippet of code shows how to use the `DatePickerTableCell` class:

```

TableColumn<Person, LocalDate> birthDateCol = ...

// Set a cell factory for birthDateCol. The date format is
// mm/dd/yyyy
// and the DatePicker is editable.
birthDateCol.setCellFactory(DatePickerTableCell.<Person>forTableC
olumn());

// Set a cell factory for birthDateCol. The date format is "Month
day, year"
// and the DatePicker is non-editable
StringConverter converter = new LocalDateStringConverter("MMMM
dd, yyyy");
birthDateCol.setCellFactory(DatePickerTableCell.<Person>forTableC
olumn(converter, false));

```

The program in Listing 13-8 uses `DatePickerTableCell` to edit data in the cells of a Birth Date column. Run the application and then double-click a cell in the Birth Date column. The cell will display a `DatePicker` control. You cannot edit the date in the `DatePicker`, as it is noneditable. You will need to select a date from the pop-up calendar.

### ***Listing 13-8.*** Using `DatePickerTableCell` to Edit a Dates in Cells

```

// CustomTableCellTest.java
package com.jdojo.control;

```

```

import com.jdojo.mvc.model.Person;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.TableColumn;
import javafx.scene.control TableView;
import javafx.scene.layout.HBox;
import java.time.LocalDate;
import javafx.stage.Stage;
import javafx.util.StringConverter;

public class CustomTableCellTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    @SuppressWarnings("unchecked")
    public void start(Stage stage) {
        TableView<Person> table = new
TableView<>(PersonTableUtil.getPersonList());

        // Make sure teh TableView is editable
        table.setEditable(true);

        // Set up teh Birth Date column to use
DatePickerTableCell
        TableColumn<Person, LocalDate> birthDateCol =
            PersonTableUtil.getBirthDateColumn();
        StringConverter converter = new
LocalDateStringConverter("MMMM dd, yyyy");
        birthDateCol.setCellFactory(
            DatePickerTableCell.<Person>forTableColumn(con
verter, false));

        table.getColumns().addAll(PersonTableUtil.getIdColum
n(),
            PersonTableUtil.getFirstNameColu
mn(),
            PersonTableUtil.getLastNameColum
n(),
            birthDateCol);

        HBox root = new HBox(table);
        root.setStyle("-fx-padding: 10;" +
                    "-fx-border-style: solid inside;" +
                    "-fx-border-width: 2;" +
                    "-fx-border-insets: 5;" +
                    "-fx-border-radius: 5;" +
                    "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using a Custom TableCell");
        stage.show();
    }
}

```

```

    }
}
}
```

## Adding and Deleting Rows in a *TableView*

Adding and deleting rows in a *TableView* are easy. Note that each row in a *TableView* is backed by an item in the items list. Adding a row is as simple as adding an item in the items list. When you add an item to the items list, a new row appears in the *TableView* at the same index as the index of the added item in the items list. If the *TableView* is sorted, it may need to be resorted after adding a new row. Call the `sort()` method of the *TableView* to resort the rows after adding a new row.

You can delete a row by removing its item from the items list. An application provides a way for the user to indicate the rows that should be deleted. Typically, the user selects one or more rows to delete. Other options are to add a Delete button to each row or to provide a Delete check box to each row. Clicking the Delete button should delete the row. Selecting the Delete check box for a row indicates that the row is marked for deletion.

The program in Listing 13-9 shows how to add and delete rows to a *TableView*. It displays a window with three sections:

- The Add Person form at the top has three fields to add person details and an Add button. Enter the details for a person and click the Add button to add a record to the *TableView*. Error checking is skipped in the code.
- In the middle, you have two buttons. One button is used to restore the default rows in the *TableView*. Another button deletes the selected rows.
- At the bottom, a *TableView* is displayed with some rows. The multirow selection is enabled. Use the Ctrl or Shift key with the mouse to select multiple rows.

### ***Listing 13-9.*** Adding and Deleting Rows in a *TableView*

```

// TableViewAddDeleteRows.java
package com.jdojo.control;

import com.jdojo.mvc.model.Person;
import javafx.application.Application;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.DatePicker;
import javafx.scene.control.Label;
```

```

import javafx.scene.control.TableView;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import java.util.Arrays;
import javafx.scene.control.SelectionMode;
import javafx.scene.layout.HBox;
import static
javafx.scene.control.TableViewSelectionModel;

public class TableViewAddDeleteRows extends Application {
    // Fields to add Person details
    private final TextField fNameField = new TextField();
    private final TextField lNameField = new TextField();
    private final DatePicker dobField = new DatePicker();

    // The TableView
    TableView<Person> table = new
TableView<>(PersonTableUtil.getPersonList());

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

    @Override
    @SuppressWarnings("unchecked")
    public void start(Stage stage) {
        // Turn on multi-row selection for the TableView
        TableViewSelectionModel<Person> tsm
= table.getSelectionModel();
        tsm.setSelectionMode(SelectionMode.MULTIPLE);

        // Add columns to the TableView
        table.getColumns().addAll(PersonTableUtil.getIdColum
n(),
                           PersonTableUtil.getFirstNameColu
mn(),
                           PersonTableUtil.getLastNameColu
n(),
                           PersonTableUtil.getBirthDateColu
mn());
    }

    GridPane newDataPane = this.getNewPersonDataPane();

    Button restoreBtn = new Button("Restore Rows");
    restoreBtn.setOnAction(e -> restoreRows());

    Button deleteBtn = new Button("Delete Selected
Rows");
    deleteBtn.setOnAction(e -> deleteSelectedRows());

    VBox root = new VBox(newDataPane, new
HBox(restoreBtn, deleteBtn), table);
    root.setSpacing(5);
}

```

```

        root.setStyle("-fx-padding: 10;" +
                      "-fx-border-style: solid inside;" +
                      "-fx-border-width: 2;" +
                      "-fx-border-insets: 5;" +
                      "-fx-border-radius: 5;" +
                      "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Adding/Deleting Rows in
a TableView");
        stage.show();
    }

    public GridPane getNewPersonDataPane() {
        GridPane pane = new GridPane();
        pane.setHgap(10);
        pane.setVgap(5);
        pane.addRow(0, new Label("First Name:"),
fNameField);
        pane.addRow(1, new Label("Last Name:"), lNameField);
        pane.addRow(2, new Label("Birth Date:"), dobField);

        Button addBtn = new Button("Add");
        addBtn.setOnAction(e -> addPerson());

        // Add the "Add" button
        pane.add(addBtn, 2, 0);

        return pane;
    }

    public void deleteSelectedRows() {
        TableViewSelectionModel<Person> tsm
= table.getSelectionModel();
        if (tsm.isEmpty()) {
            System.out.println("Please select a row to
delete.");
            return;
        }

        // Get all selected row indices in an array
        ObservableList<Integer> list
= tsm.getSelectedIndices();
        Integer[] selectedIndices = new
Integer[list.size()];
        selectedIndices = list.toArray(selectedIndices);

        // Sort the array
        Arrays.sort(selectedIndices);

        // Delete rows (last to first)
        for(int i = selectedIndices.length - 1; i >= 0; i--)
{
            tsm.clearSelection(selectedIndices[i].intValue
}
    }
}

```

```
(() );
        table.getItems().remove(selectedIndices[i].int
Value());
    }
}

public void restoreRows() {
    table.getItems().clear();
    table.getItems().addAll(PersonTableUtil.getPersonLis
t());
}
}

public Person getPerson() {
    return new Person(fNameField.getText(),
        lNameField.getText(),
        dobField.getValue());
}

public void addPerson() {
    Person p = getPerson();
    table.getItems().add(p);
    clearFields();
}

public void clearFields() {
    fNameField.setText(null);
    lNameField.setText(null);
    dobField.setValue(null);
}
}
```

Most of the logic in the code is simple.

The `deleteSelectedRows()` method implements the logic to delete the selected rows. When you remove an item from the items list, the selection model does not remove its index. Suppose the first row is selected. If you remove the first item from the items list, the second row, which becomes the first row, is selected. To make sure that this does not happen, you clear the selection for the row before you remove it from the items list. You delete rows from last to first (higher index to lower index) because when you delete an item from the list, all of the items after the deleted items will have different indices. Suppose you have selected rows at indices 1 and 2. Deleting a row at index 1 first changes the index of the index 2 to 1. Performing deletion from last to first takes care of this issue.

## Scrolling in a *TableView*

`TableView` automatically provides vertical and horizontal scrollbars when rows or columns fall beyond the available space. Users can use the scrollbars to scroll to a specific row or column. Sometimes you need programmatic support for scrolling. For example, when you append a row to a `TableView`, you may want the row visible to the user by

scrolling it to the view. The `TableView` class contains four methods that can be used to scroll to a specific row or column:

- `scrollTo(int rowIndex)`
- `scrollTo(S item)`
- `scrollToColumn(TableColumn<S, ?> column)`
- `scrollToColumnIndex(int columnIndex)`

The `scrollTo()` method scrolls the row with the specified index or item to the view.

The `scrollToColumn()` and `scrollToColumnIndex()` methods scroll to the specified column and `columnIndex`, respectively.

`TableView` fires a `ScrollToEvent` when there is a request to scroll to a row or column using one of the above-mentioned scrolling methods. The `ScrollToEvent` class contains a `getScrollTarget()` method that returns the row index or the column reference depending on the scroll type:

```
TableView<Person> table = ...

// Add a ScrollToEvent for row scrolling
table.setOnScrollTo(e -> {
    int rowIndex = e.getScrollTarget();
    System.out.println("Scrolled to row " + rowIndex);
});

// Add a ScrollToEvent for column scrolling
table.setOnScrollToColumn(e -> {
    TableColumn<Person, ?> column = e.getScrollTarget();
    System.out.println("Scrolled to column "
+ column.getText());
});
```

**Tip** The `ScrollToEvent` is not fired when the user scrolls through the rows and columns. It is fired when you call one of the four scrolling-related methods of the `TableView` class.

## Resizing a `TableColumn`

Whether a `TableColumn` is resizable by the user is specified by its `resizable` property. By default, a `TableColumn` is resizable. How a column in a `TableView` is resized is specified by the `columnResizePolicy` property of the `TableView`. The property is a `Callback` object. Its `call()` method takes an object of the `ResizeFeatures` class, which is a static inner class of the `TableView` class. The `ResizeFeatures` object encapsulates the delta by which the column is resized, the `TableColumn` being resized, and the `TableView`. The `call()` method returns `true` if the column

was resized by the delta amount successfully. Otherwise, it returns `false`.

The `TableView` class provides two built-in resize policies as constants:

- `CONSTRAINED_RESIZE_POLICY`
- `UNCONSTRAINED_RESIZE_POLICY`

`CONSTRAINED_RESIZE_POLICY` ensures that the sum of the width of all visible leaf columns is equal to the width of the `TableView`. Resizing a column adjusts the width of all columns to the right of the resized column. When the column width is increased, the width of the rightmost column is decreased up to its minimum width. If the increased width is still not compensated, the width of the second rightmost column is decreased up to its minimum width and so on. When all columns to the right have their minimum widths, the column width cannot be increased any more. The same rule applies in the opposite direction when a column is resized to decrease its width.

When the width of a column is increased, `UNCONSTRAINED_RESIZE_POLICY` shifts all columns to its right by the amount the width is increased. When the width is decreased, columns to the right are shifted to the left by the same amount. If a column has nested columns, resizing the column evenly distributes the delta among the immediate children columns. This is the default column-resize policy for a `TableView`:

```
TableView<Person> table = ...;

// Set the column resize policy to constrained resize policy
table.setColumnResizePolicy(TableView.CONSTRAINED_RESIZE_POLICY);
```

You can also create a custom column resize policy. The following snippet of code will serve as a template. You will need to write the logic to consume the delta, which is the difference between the new and old width of the column:

```
TableView<Person> table = new
TableView<>(PersonTableUtil.getPersonList());
table.setColumnResizePolicy(resizeFeatures -> {
    boolean consumedDelta = false; double delta
= resizeFeatures.getDelta();
    TableColumn<Person, ?> column = resizeFeatures.getColumn();
    TableView<Person> tableView = resizeFeatures.getTable();

    // Adjust the delta here...

    return consumedDelta;
});
```

You can disable column resizing by setting a trivial callback that does nothing. Its `call()` simply returns `true` indicating that it has consumed the delta:

```
// Disable column resizing
table.setColumnResizePolicy(resizeFeatures -> true);
```

## **Styling a *TableView* with CSS**

You can style a `TableView` and all its parts, for example, column headers, cells, placeholder, and so forth. Applying a CSS to `TableView` is very complex and broad in scope. This section covers a brief overview of CSS styling for `TableView`. The default CSS style-class name for a `TableView` is `table-view`. The default CSS style-classes for a cell, a row, and a column header are `table-cell`, `table-row-cell`, and `column-header`, respectively:

```
/* Set the font for the cells */
.table-row-cell {
    -fx-font-size: 10pt;
    -fx-font-family: Arial;
}

/* Set the font size and text color for column headers */
.table-view .column-header .label{
    -fx-font-size: 10pt;
    -fx-text-fill: blue;
}
```

`TableView` supports the following CSS pseudo-classes:

- `cell-selection`
- `row-selection`
- `constrained-resize`

The `cell-selection` pseudo-class is applied when the cell-level selection is enabled, whereas the `row-selection` pseudo-class is applied for row-level selection. The `constrained-resize` pseudo-class is applied when the column resize policy is `CONSTRAINED_RESIZE_POLICY`.

Alternate rows in a `TableView` are highlighted by default. The following code removes the alternate row highlighting. It sets the white background color for all rows:

```
.table-row-cell {
    -fx-background-color: white;
}

.table-row-cell .table-cell {
    -fx-border-width: 0.25px;
```

```
-fx-border-color: transparent gray gray transparent;  
}
```

TableView shows empty rows to fill its available height. The following code removes the empty rows. In fact, it makes them appear as removed:

```
.table-row-cell:empty {  
    -fx-background-color: transparent;  
}  
  
.table-row-cell:empty .table-cell {  
    -fx-border-width: 0px;  
}
```

TableView contains several substructures that can be styled separately:

- column-resize-line
- column-overlay
- placeholder
- column-header-background

The column-resize-line substructure is a Region and is shown when the user tries to resize a column. The column-overlay substructure is a Region and is shown as an overlay for the column being moved. The placeholder substructure is a StackPane and is shown when the TableView does not have columns or data, as in the following code:

```
/* Make the text in the placeholder red and bold */  
.table-view .placeholder .label {  
    -fx-text-fill: red;  
    -fx-font-weight: bold;  
}
```

The column-header-background substructure is a StackPane, and it is the area behind the column headers. It contains several substructures. Its filler substructure, which is a Region, is the area between the rightmost column and the right edge of the TableView in the header area. Its show-hide-columns-button substructure, which is a StackPane, is the area that shows the menu button to display the list of columns to show and hide. Please refer to the modena.css file and the *JavaFX CSS Reference Guide* for a complete list of properties of TableView that can be styled. The following code sets the filler background to white:

```
/* Set the filler background to white*/  
.table-view .column-header-background .filler {  
    -fx-background-color: white;  
}
```

## Summary

`TableView` is a control that is used to display and edit data in a tabular form. A `TableView` consists of rows and columns. The intersection of a row and a column is called a cell. Cells contain the data values. Columns have headers that describe the type of data they contain. Columns can be nested. Resizing and sorting of column data have built-in support. The following classes are used to work with

a `TableView` control: `TableView`, `TableColumn`, `TableRow`, `TableCell`, `TablePosition`, `TableView.TableViewFocusModel`, and `TableView.TableViewSelectionModel`. The `TableView` class represents a `TableView` control. The `TableColumn` class represents a column in a `TableView`. Typically, a `TableView` contains multiple instances of `TableColumn`. A `TableColumn` consists of cells, which are instances of `TableCell` class. A `TableColumn` is responsible for displaying and editing the data in its cells. A `TableColumn` has a header that can display header text, a graphic, or both. You can have a context menu for a `TableColumn`, which is displayed when the user right-clicks inside the column header. Use the `contextMenu` property to set a context menu.

The `TableRow` class inherits from the `IndexedCell` class. An instance of `TableRow` represents a row in a `TableView`. You almost never use this class in your application unless you want to provide a customized implementation for rows. Typically, you customize cells, not rows.

An instance of the `TableCell` class represents a cell in a `TableView`. Cells are highly customizable. They display data from the underlying data model for the `TableView`. They are capable of displaying data as well as graphics. Cells in a row of a `TableView` contain data related to an item such as a person, a book, and so forth. Data for some cells in a row may come directly from the attributes of the item or they may be computed.

`TableView` has an `items` property of the `ObservableList<S>` type. The generic type `S` is the same as the generic type of the `TableView`. It is the data model for the `TableView`. Each element in the `items` list represents a row in the `TableView`. Adding a new item to the `items` list adds a new row to the `TableView`. Deleting an item from the `items` list deletes the corresponding row from the `TableView`.

The `TableColumn`, `TableRow`, and `TableCell` classes contain a `tableView` property that holds the reference of the `TableView` that

contains them. The `tableView` property contains null when the `TableColumn` does not belong to a `TableView`.

A `TablePosition` represents the position of a cell. Its `getRow()` and `getColumn()` methods return the indices of rows and columns, respectively, to which the cell belongs.

The `TableViewFocusModel` class is an inner static class of the `TableView` class. It represents the focus model for the `TableView` to manage focus for rows and cells.

The `TableViewSelectionModel` class is an inner static class of the `TableView` class. It represents the selection model for the `TableView` to manage selection for rows and cells.

By default, all columns in a `TableView` are visible.

The  `TableColumn` class has a `visible` property to set the visibility of a column. If you turn off the visibility of a parent column, a column with nested columns, all of its nested columns will be invisible.

You can rearrange columns in a `TableView` in two ways: by dragging and dropping columns to a different position or by changing their positions in the observable list of returned by the `getColumns()` method of the `TableView` class. The first option is available by default.

`TableView` has built-in support for sorting data in columns. By default, it allows users to sort data by clicking column headers. It also supports sorting data programmatically. You can also disable sorting for a column or all columns in a `TableView`.

`TableView` supports customization at several levels. It lets you customize the rendering of columns, for example, you can display data in a column using a check box, a combo box, or a `TextField`. You can also style a `TableView` using CSS.

The next chapter will discuss the tree view control that is used to work with data representing a tree-like hierarchical structure.

## CHAPTER 14



### Understanding *TreeView*

---

In this chapter, you will learn:

- What a *TreeView* is
- How to create a *TreeView*
- How to hide the root node of a *TreeView*
- What a *TreeItem* is and how to handle *TreeItem* events in a *TreeView*
- How to customize cells in a *TreeView*
- How to edit data in a *TreeView*
- How to load a *TreeItem* in a *TreeView* on demand
- About the selection model of the *TreeView*
- How to style a *TreeView* using CSS

#### What Is a *TreeView* ?

A *TreeView* is a control that displays hierarchical data in a tree-like structure, as shown in Figure 14-1. You can think of a *TreeView* as displaying a tree upside down—the root of the tree being at the top. Each item in a *TreeView* is an instance of the *TreeItem* class. *TreeItems* form parent-child relationships. In Figure 14-1, Departments, IS, and Doug Dyer are instances of a *TreeItem*.

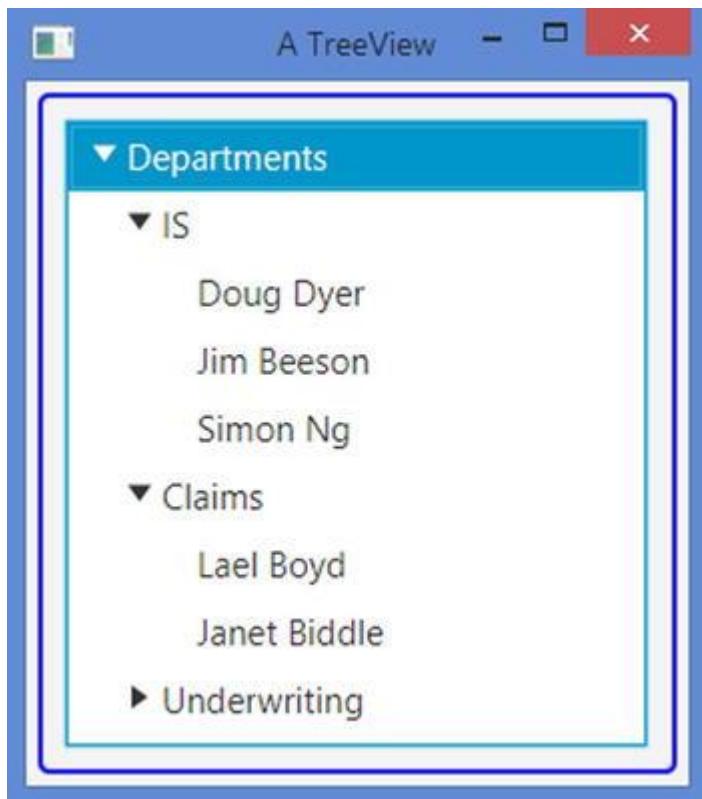


Figure 14-1. A window with a `TreeView` control

A `TreeItem` is also referred to as a *node*. The `TreeItem` class does not inherit from the `Node` class. Therefore, a `TreeItem` is not a JavaFX `Node` and it cannot be added to a scene graph.

A `TreeItem` is categorized as a *branch* or *leaf node*. If a `TreeItem` contains other instances of `TreeItem`, which are called its children, it is called a branch node. Otherwise, it is called a leaf node. In Figure 14-1, Departments, IS, and Claims are examples of branch nodes, whereas Doug Dyer and Lael Boyd are examples of leaf nodes. Notice that leaf nodes are those that occur at the tips of the tree hierarchy. A leaf node has a parent but no children. A branch node has a parent as well as children, except a special branch node, which is called the *root node*. The root node has no parent, but children only, and it is the first node in the `TreeView`. Departments is the root node in Figure 14-1.

A branch node can be in an *expanded* or *collapsed* state. In Figure 14-1, the Departments, IS, and Claims nodes are in the expanded state, whereas the Underwriting node is in the collapsed state. A triangle, which is called a *disclosure node*, is used to show the expanded and collapsed state of a branch node.

A `TreeItem` serves as the data model in a `TreeView`. Each `TreeItem` uses an instance of the `TreeCell` class to render its

value. A TreeCell in a TreeView can be customized using a cell factory. By default, a TreeCell is not editable.

TreeView is a virtualized control. It creates only as many instances of TreeCell as needed to display the items for its current height. Cells are recycled as you scroll through items. Virtualization makes it possible to use TreeView for viewing very large number of items without using a large amount of memory. Note, however, that loading TreeItems always takes memory. Virtualization helps only in viewing the items by recycling the cells used in viewing them.

## **Creating a TreeView**

An instance of the TreeView<T> class represents a TreeView control. The TreeView class takes a generic type, which is the type of the value contained in its TreeItems. The default constructor creates an empty TreeView:

```
// Create an empty TreeView whose TreeItems value type is String
TreeView<String> treeView = new TreeView<>();
```

Another constructor creates a TreeView with the root node:

```
// Create the root TreeItem
TreeItem<String> depts = new TreeItem<String>("Departments");

// Create a TreeView with depts as its root item
TreeView<String> treeView = new TreeView<>(depts);
```

The TreeView class contains a root property. Its type is TreeItem<T> and it represents the root node. You can create an empty TreeView and set its root node later using the setRoot() method:

```
// Create an empty TreeView whose TreeItems value is String
TreeView<String> treeView = new TreeView<>();

...
// Set the root node
treeView.setRoot(depts);
```

A TreeItem stores all its children in an ObservableList. The getChildren() method returns the reference of the list. The following snippet of code adds three children TreeItems to the root:

```
// Create children TreeItems for the root
TreeItem<String> isDept = new TreeItem<String>("IS");
TreeItem<String> claimsDept = new TreeItem<String>("Claims");
TreeItem<String> underwritingDept = new
TreeItem<String>("Underwriting");

// Add children to the root
depts.getChildren().addAll(isDept, claimsDept, underwritingDept);
```

You can use the above logic to build a TreeView and add as many instances of TreeItem as you like. Notice that you add only the root

node to the TreeView. All other nodes are added to the root node and its children.

You can use a TreeView with the same TreeItems several times. Let's look at the code to build a TreeView once and then reuse it. Listing 14-1 is a utility class. Its getTreeView() method shows how to create and populate a TreeView. It returns the reference of the TreeView. When you need a TreeView in an example, you will use this method.

### ***Listing 14-1.*** A TreeView Utility Class that Builds a TreeView

```
// TreeViewUtil.java
package com.jdojo.control;

import javafx.scene.control.TreeItem;
import javafx.scene.control.TreeView;

public class TreeViewUtil {
    public static TreeView<String> getTreeView() {
        TreeItem<String> depts = new
TreeItem<>("Departments");

        // Add items to depts
        TreeItem<String> isDept = new TreeItem<String>("IS");
        TreeItem<String> claimsDept = new
TreeItem<String>("Claims");
        TreeItem<String> underwritingDept = new
TreeItem<String>("Underwriting");
        depts.getChildren().addAll(isDept, claimsDept,
underwritingDept);

        // Add employees for each dept
        isDept.getChildren().addAll(new
TreeItem<String>("Doug Dyer"),
                                new TreeItem<String>("Jim
Beeson"),
                                new TreeItem<String>("Simon
Ng"));

        claimsDept.getChildren().addAll(new
TreeItem<String>("Lael Boyd"),
                                new TreeItem<String>("Janet
Biddle"));

        underwritingDept.getChildren().addAll(new
TreeItem<String>("Ken McEwen"),
                                new
TreeItem<String>("Ken Mann"),
                                new
TreeItem<String>("Lola Ng"));

        // Create a TreeView with depts as its root item
    }
}
```

```

        TreeView<String> treeView = new TreeView<>(dept);
        return treeView;
    }
}

```

The program Listing 14-2 shows a `TreeView` control in a window. When you run the program, all nodes are collapsed, which is the default behavior for a `TreeView`. You will need to click the disclosure node (the triangle) for the root node to expand it and view its children. Repeat this to expand other nodes. Clicking the disclosure node for an expanded node hides its children.

**Tip** By default, a node is in the collapsed state. Call the `setExpanded(true)` method of a `TreeItem` to expand a node.

### ***Listing 14-2.*** Creating a `TreeView` Control

```

// TreeViewTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.TreeView;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class TreeViewTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        TreeView<String> treeView
= TreeViewUtil.getTreeView();
        HBox root = new HBox(treeView);
        root.setStyle("-fx-padding: 10;" +
                     "-fx-border-style: solid inside;" +
                     "-fx-border-width: 2;" +
                     "-fx-border-insets: 5;" +
                     "-fx-border-radius: 5;" +
                     "-fx-border-color: blue;" );
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Creating a TreeView");
        stage.show();
    }
}

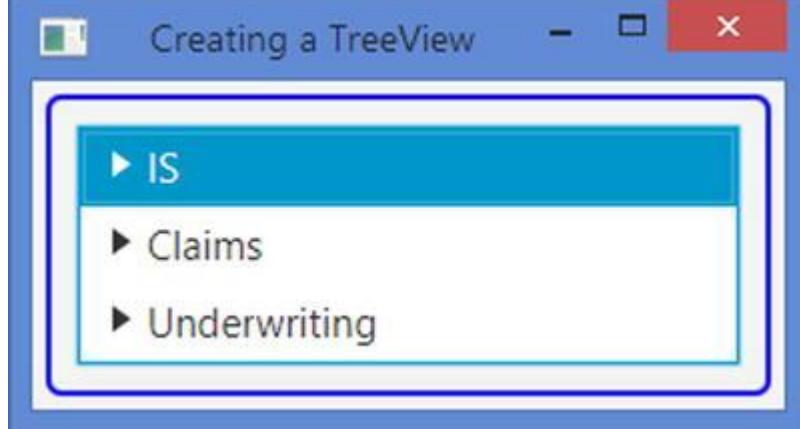
```

### **Hiding the Root Node**

In a `TreeView`, you can hide the root node by setting the value for its `showRoot` property to false. By default, the root node is visible. Call `setShowRoot(false)` of the `TreeView` to hide the root node. Hiding root node makes traversing the `TreeView` a little easier as the user has one less level of indentation to traverse. Hiding the root node shows its child nodes at the first level. The following snippet of code will display a `TreeView`, as shown in Figure 14-2.

```
TreeView<String> treeView = TreeViewUtil.getTreeView();

// Hide the root node
treeView.setShowRoot(false);
```



**Figure 14-2.** A `TreeView` with its root node hidden

### Understanding the `TreeItem`

A `TreeItem` supplies the data for a node. It has the following properties:

- `expanded`
- `graphic`
- `leaf`
- `parent`
- `value`

The `expanded` property indicates whether a `TreeItem` is expanded. It is true if the `TreeItem` is in the expanded state. Otherwise, it is false. You can expand a node using the `setExpanded(true)` method.

A `TreeItem` may optionally contain an icon represented by its `graphic` property. The type of the `graphic` property is `Node`, and therefore, you can use any node for the `graphic` property. Typically, a 16-by-16 image is used.

The `leaf` property indicates whether a `TreeItem` has children. It is true if the `TreeItem` has no children. Otherwise, it is false. It is a read-only property.

Every TreeItem in a TreeView, except the root TreeItem, has a parent TreeItem. The parent property is a read-only property that contains the parent of the TreeItem.

The value property stores the application-specific data for a TreeItem. Its type is the same as the generic type of the TreeItem class.

You can get the ObservableList of children of a TreeItem using the getChildren() method. You can traverse the tree up or down from a TreeItem using the getParent() and getChildren() methods recursively.

The TreeItem class provides several constructors to create an empty TreeItem, a TreeItem with a value, and a TreeItem with a value and graphic:

```
// Create an empty TreeItem and set the value
TreeItem<String> emptyItem = new TreeItem<>();
emptyItem.setValue("Departments");

// A TreeItem with a value
TreeItem<String> item2 = new TreeItem<>("Departments");

// A TreeItem with a value and an icon
ImageIcon icon = ...
TreeItem<String> item3 = new TreeItem<>("Departments", icon);
```

## Handling *TreeItem* Events

A TreeItem fires events as it is modified, for example, by adding or removing children or expanding or collapsing. An instance of the TreeModificationEvent class, which is a static inner class of the TreeItem class, represents all kinds of modification events.

Different types of events are represented by different event types. It is a little strange that the TreeItem class does not contain constants for those event types. Rather, it contains static methods that return those event types. For example,

the TreeItem.branchCollapsedEvent() static method returns the event type of the event that is fired when a TreeItem is collapsed.

Event types are arranged in a hierarchy.

The TreeNotification event type is at the top of the hierarchy. It is the parent of all event types for TreeItem. You can add an event handler for this event type to a TreeItem and it will listen for all event types for a TreeItem. The following three event types are the direct subtypes of the TreeNotification event type:

- ValueChanged

- GraphicChanged
- TreeItemCountChange

The ValueChanged and GraphicChanged event types are fired when the value and graphic properties, respectively, of the TreeItem change. The TreeItemCountChange event type is fired when the TreeItem is expanded, collapsed, or its children list is changed. It has three subtypes to handle the specific events:

- BranchExpanded
- BranchCollapsed
- ChildrenModification

You should add event handlers for specific type of events for better performance. When an event occurs on a TreeItem, all the registered listeners are called. The event bubbles up the TreeItem chain following the parent of the TreeItem until the root TreeItem is reached. Therefore, if you want to handle a specific event on all TreeItems, add an event handler only to the root TreeItem. The following snippet of code creates a TreeView with a root node. It adds BranchExpanded and BranchCollapsed event handlers to the root node. These event handlers will be called whenever any branch in the TreeView is expanded or collapsed. The handlers print a message on the standard output about the node being expanded or collapsed.

```
TreeItem<String> depts = new TreeItem<>("Departments");
TreeView<String> treeView = new TreeView<>(depts);

// Add BranchExpended event handler
depts.addEventListener(TreeItem.<String>branchExpandedEvent(),
    e -> System.out.println("Node expanded: "
+ e.getSource().getValue()));

// Add BranchCollapsed event handler
depts.addEventListener(TreeItem.<String>branchCollapsedEvent(),
    e -> System.out.println("Node collapsed: "
+ e.getSource().getValue()));
```

## Adding and Removing Nodes

Adding and removing TreeItems is as easy as adding or removing them in the children list of their parents. Notice that the root node does not have a parent. To delete the root node, you need to set the root property of the TreeView to null.

The program in Listing 14-3 shows how to add and remove nodes in a TreeView. A TreeView with a root node is displayed in the left side of

the window. The right side displays a `TextField` and an `Add` button. Enter a text and click the `Add` button; a new node will be added under the selected node. Click the `Remove Selected Item` button to remove the selected node from the `TreeView`. At the bottom of the window, a detailed message log is displayed in a `TextArea`. The program also shows how to handle `TreeItem` events.

### ***Listing 14-3.*** Adding and Deleting Nodes in a `TreeView`

```
// TreeItemAddDeleteTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TreeItem;
import javafx.scene.control.TreeView;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;
import javafx.scene.control.TextArea;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;

public class TreeItemAddDeleteTest extends Application {
    private final TreeView<String> treeView = new TreeView<>();
    private final TextArea msgLogFld = new TextArea();

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

    @Override
    public void start(Stage stage) {
        // Select the root node
        treeView.getSelectionModel().selectFirst();

        // Create the root node and adds event handler to it
        TreeItem<String> depts = new
TreeItem<>("Departments");
        depts.addEventHandler(TreeItem.<String>branchExpanded
Event(),
                           this::branchExpanded);
        depts.addEventHandler(TreeItem.<String>branchCollapse
Event(),
                           this::branchCollapsed);
        depts.addEventHandler(TreeItem.<String>childrenModifi
cationEvent(),
                           this::childrenModification);

        // Set the root node for the TreeViww
        treeView.setRoot(depts);
    }
}
```

```

VBox rightPane = getRightPane();

HBox root = new HBox(treeView, rightPane);
root.setSpacing(20);
root.setStyle("-fx-padding: 10;" +
              "-fx-border-style: solid inside;" +
              "-fx-border-width: 2;" +
              "-fx-border-insets: 5;" +
              "-fx-border-radius: 5;" +
              "-fx-border-color: blue;");

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("Creating a TreeView");
stage.show();
}

public VBox getRightPane() {
    TextField itemFld = new TextField();

    Button addItemBtn = new Button("Add");
    addItemBtn.setOnAction(e ->
this.addItem(itemFld.getText()));

    Button removeItemBtn = new Button("Remove Selected
Item");
    removeItemBtn.setOnAction(e -> this.removeItem());

    msgLogFld.setPrefRowCount(15);
    msgLogFld.setPrefColumnCount(25);
    VBox box = new VBox(new Label("Select an item to add
to or remove."),
                      new HBox(new Label("Item:"), itemFld,
addItemBtn),
                      removeItemBtn,
                      new Label("Message Log:"), msgLogFld);
    box.setSpacing(10);
    return box;
}

public void addItem(String value) {
    if (value == null || value.trim().equals("")) {
        this.logMsg("Item cannot be empty.");
        return;
    }

    TreeItem<String> parent
= treeView.getSelectionModel().getSelectedItem();
    if (parent == null) {
        this.logMsg("Select a node to add this item
to.");
        return;
    }
}

```

```

        // Check for duplicate
        for(TreeItem<String> child : parent.getChildren()) {
            if (child.getValue().equals(value)) {
                this.logMsg(value + " already exists under
" + parent.getValue());
                return;
            }
        }

        TreeItem<String> newItem = new
TreeItem<String>(value);
        parent.getChildren().add(newItem);
        if (!parent.isExpanded()) {
            parent.setExpanded(true);
        }
    }

    public void removeItem() {
        TreeItem<String> item
= treeView.getSelectionModel().getSelectedItem();
        if (item == null) {
            this.logMsg("Select a node to remove.");
            return;
        }

        TreeItem<String> parent = item.getParent();
        if (parent == null) {
            this.logMsg("Cannot remove the root node.");
        } else {
            parent.getChildren().remove(item);
        }
    }

    public void
branchExpanded(TreeItem.TreeModificationEvent<String> e) {
    String nodeValue = e.getSource().getValue();
    this.logMsg("Event: " + nodeValue + " expanded.");
}

    public void
branchCollapsed(TreeItem.TreeModificationEvent<String> e) {
    String nodeValue = e.getSource().getValue();
    this.logMsg("Event: " + nodeValue + " collapsed.");
}

    public void
childrenModification(TreeItem.TreeModificationEvent<String> e) {
        if (e.wasAdded()) {
            for(TreeItem<String> item
: e.getAddedChildren()) {
                this.logMsg("Event: " + item.getValue()
+ " has been added.");
            }
        }
    }
}

```

```

        }

        if (e.wasRemoved()) {
            for(TreeItem<String> item
: e.getRemovedChildren()) {
                this.logMsg("Event: " + item.getValue()
+ " has been removed.");
            }
        }
    }

    public void logMsg(String msg) {
        this.msgLogFld.appendText(msg + "\n");
    }
}

```

## Customizing Cells in a TreeView

TreeView uses a TreeCell to render a TreeItem. A TreeCell is an IndexedCell. You can visualize items in a TreeView from top to bottom arranged in rows. Each row has exactly one item. Each item is given a row index. The first item, which is the root item, has the index of zero. The row indices are given only to the visible items. TreeView contains a read-only expandedItemCount property that is the number of visible items. Use the getExpandedItemCount() method to get the number of visible items. If a node above an item is expanded or collapsed, the index of the item changes to reflect new visible items. The index of a TreeCell in a TreeView and the row index of an item are the same. Use the getIndex() method of the TreeCell or the getRow(TreeItem<T> item) method of the TreeView to get the row index of an item.

A TreeCell is a Labeled control. By default, it uses the following rules to render its TreeItem: If the value in the TreeItem is an instance of the Node class, the value is displayed using the graphic property of the cell. Otherwise, the toString() method of the value is called and the returned string is displayed using the text property of the cell.

You can take full control of how a TreeCell renders its TreeItem by providing a cell factory for the TreeView. The cellFactory property is a Callback instance, which takes the TreeView as an argument and returns a TreeCell. The following snippet of code shows how to sets a cell factory to a TreeView:

```

TreeView<String> treeView = new TreeView<>();
...
// Set a cell factory to prepend the row index to the TreeItem

```

```

value
treeView.setCellFactory( (TreeView<String> tv) -> {
    TreeCell<String> cell = new TreeCell<String>() {
        @Override
        public void updateItem(String item, boolean empty) {
            super.updateItem(item, empty);
            /* Logic to render the cell goes here */
        }
    };
    return cell;
});

```

**Listing 14-4** has a complete program that uses a cell factory for a TreeView. The cell displays the index of the cell followed with the value of the TreeItem. The program displays a window as shown in Figure 14-3.

### **Listing 14-4.** Using a Cell Factory for a TreeView

```

// TreeViewCellFactory.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.TreeCell;
import javafx.scene.control.TreeView;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class TreeViewCellFactory extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        TreeView<String> treeView
= TreeViewUtil.getTreeView();

        // Set a cell factory to prepend the row index to the
        // TreeItem value
        treeView.setCellFactory( (TreeView<String> tv) -> {
            TreeCell<String> cell = new TreeCell<String>() {
                @Override
                public void updateItem(String item,
boolean empty) {
                    super.updateItem(item, empty);
                    if (empty) {
                        this.setText(null);
                        this.setGraphic(null);
                    }
                    else {
                        String value =

```

th

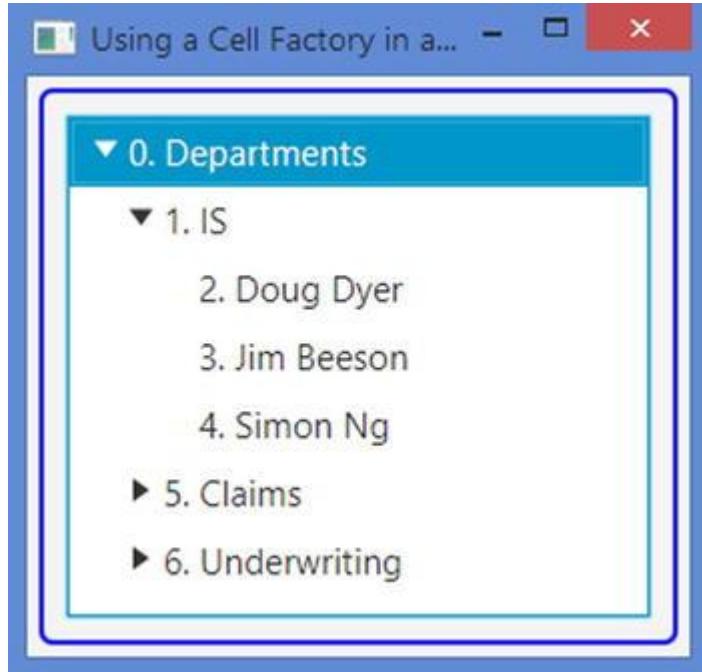
```

is.getTreeItem().getValue();
        this.setText(
            th
is.getIndex() + ". " + value);
    }
}
return cell;
});

HBox root = new HBox(treeView);
root.setSpacing(20);
root.setStyle("-fx-padding: 10;" +
    "-fx-border-style: solid inside;" +
    "-fx-border-width: 2;" +
    "-fx-border-insets: 5;" +
    "-fx-border-radius: 5;" +
    "-fx-border-color: blue;");

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("Using a Cell Factory in a TreeView");
stage.show();
}
}

```



**Figure 14-3.** A TreeView showing the row index of its TreeItems

## Editing Data in a TreeView

A cell in a TreeView can be editable. An editable cell may switch between editing and nonediting mode. In editing mode, cell data can be modified by the user. For a cell to enter editing mode, the TreeView must be editable. TreeView has an editable property,

which can be set to true using the `setEditable(true)` method, as shown in the following code. By default, `TreeView` is not editable.

```
TreeView<String> treeView = ...  
treeView.setEditable(true);
```

`TreeView` supports three types of events:

- `onEditStart`
- `onEditCommit`
- `onEditCancel`

The `onEditStart` event is fired when a cell enters editing mode.

The `onEditCommit` event is fired when the user successfully commits the editing, for example, by pressing the Enter key in a `TextField`.

The `onEditCancel` event is fired when the user cancels the editing, for example, by pressing the Esc key in a `TextField`. The events are represented by an object of the `TreeView.EditEvent` class. The event object encapsulates the old and new values in the cell, the `TreeItem` being edited, and the reference of the `TreeView`. Use one of the methods of the `EditEvent` class to get these values.

Creating a `TreeView` does not let you edit its cells. Cell-editing capability is provided through specialized implementations of the `TreeCell` class. The JavaFX library provides some of these implementations. Set the cell factory for a `TreeView`, which is a `Callback` object, to use one of the following implementations of the `TreeCell` to make cells in a `TreeView` editable:

- `CheckBoxTreeCell`
- `ChoiceBoxTreeCell`
- `ComboBoxTreeCell`
- `TextFieldTreeCell`

`TreeView` has a read-only `editingItem` property that contains the reference of the `TreeItem` being edited. It is `null` when no `TreeItem` is being edited.

## Editing Data using a Check Box

A `CheckBoxTreeCell` renders a check box in the cell. The following snippet of code sets the cell factory for a `TreeView` to use `CheckBoxTreeCell`:

```
TreeView<String> treeView = new TreeView<>();  
  
// Set a cell factory to use TextFieldTreeCell  
treeView.setCellFactory(CheckBoxTreeCell.<String>forTreeView());
```

The above code will draw a check box in each cell of the `TreeView`, which can be selected and unselected. However, there is no way to know whether the check box for a `TreeItem` is selected or unselected because the `TreeItem` class does not provide access to the `CheckBox` state. `CheckBoxTreeItem` is a specialized implementation of `TreeItem` that should be used when you want to use `CheckBoxTreeCell`. It provides access to the selected state of the check box. It contains three boolean properties:

- `independent`
- `indeterminate`
- `selected`

The `independent` property represents the independent state of the `CheckBoxTreeItem`. By default, it is false. When a `CheckBoxTreeItem` is dependent, selecting or unselecting it affects the selected state of its children and its parent. For example, selecting or unselecting a dependent parent node selects or unselects all its children. If some, but not all, children are selected, a dependent parent node will be in an indeterminate state. If all children are selected, a dependent parent node will be selected. If all children are unselected, a dependent parent node will be unselected. The selected state of an independent `CheckBoxTreeItem` does not affect its parent and children.

**Tip** The selection of an independent `CheckBoxTreeItem` does not affect its parent and children. However, the reverse is not true. That is, if a dependent parent is selected or unselected, all its children, dependent as well as independent, are selected or unselected.

The `indeterminate` property specifies the indeterminate state of the check box for the item. The `selected` property specifies the selected property of the check box for the item. Use the `isIndeterminate()` and `isSelected()` methods to determine the state of the check box of a `CheckBoxTreeItem`.

The program in Listing 14-5 shows how to use a cell factory to render a check box in each cell with a `CheckBoxTreeItem`. The initial part of the program is very similar to the first example you had for the `TreeView`. The only difference is that, this time, you have used `CheckBoxTreeItem` instead of a `TreeItem`. You have made the `Claims` item independent. That is, selecting and unselecting the `Claims` item does not affect the state of its parent and children. Select and unselect different items to see this effect.

**Listing 14-5.** Using CheckBoxTreeItem with a Check Box in a TreeCell

```
// TreeViewCheckBoxTest.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.CheckBoxTreeItem;
import javafx.scene.control.TreeView;
import javafx.scene.control.cell.CheckBoxTreeCell;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class TreeViewCheckBoxTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    @SuppressWarnings("unchecked")
    public void start(Stage stage) {
        CheckBoxTreeItem<String> depts =
            new CheckBoxTreeItem<>("Departments");

        // Add items to depts
        CheckBoxTreeItem<String> isDept = new
        CheckBoxTreeItem<>("IS");
        CheckBoxTreeItem<String> claimsDept =
            new CheckBoxTreeItem<>("Claims");
        CheckBoxTreeItem<String> underwritingDept =
            new CheckBoxTreeItem<>("Underwriting");
        depts.getChildren().addAll(isDept, claimsDept,
        underwritingDept);

        // Add employees for each dept
        isDept.getChildren().addAll(new
        CheckBoxTreeItem<String>("Doug Dyer"),
            new
        CheckBoxTreeItem<String>("Jim Beeson"),
            new
        CheckBoxTreeItem<String>("Simon Ng"));

        claimsDept.getChildren().addAll(
            new CheckBoxTreeItem<String>("Lael
        Boyd"),
            new
        CheckBoxTreeItem<String>("Janet Biddle"));

        underwritingDept.getChildren().addAll(
            new CheckBoxTreeItem<String>("Ken
        McEwen"),
            new CheckBoxTreeItem<String>("Ken
        Mann"));
    }
}
```

```

        new CheckBoxTreeItem<String>("Lola
Ng")) ;

        // Make the claimsDept item independent
        claimsDept.setIndependent(true);

        // Create a TreeView with depts as its root item
        TreeView<String> treeView = new TreeView<>(depts);

        // Set the cell factory to draw a CheckBox in cells
        treeView.setCellFactory(CheckBoxTreeCell.<String>forT
reeView()));

        HBox root = new HBox(treeView);
        root.setSpacing(20);
        root.setStyle("-fx-padding: 10;" +
                      "-fx-border-style: solid inside;" +
                      "-fx-border-width: 2;" +
                      "-fx-border-insets: 5;" +
                      "-fx-border-radius: 5;" +
                      "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using CheckBoxTreeItem");
        stage.show();
    }
}
}

```

## Editing Data Using a Choice Box

A `ChoiceBoxTreeCell` is rendered as a `Label` in nonediting mode and as a choice box in editing mode. Its `forTreeView()` static method returns a cell factory. The method is overloaded. You need to pass the list of items to be shown in the choice box. If the `toString()` method of the item does not return a user-friendly string for the user, use a string converter. The following snippet of code sets a cell factory for a `TreeView` that will use instances of `ChoiceBoxTreeCell` to render `TreeItems`. In editing mode, the choice box will display three items: **Item-1**, **Item-2**, and **Item-3**.

```

TreeView<String> treeView = new TreeView<>();
treeView.setCellFactory(ChoiceBoxTreeCell.<String>forTreeView("It
em-1", "Item-2", "Item-3"));

```

When an item is selected in the choice box, the item is set to the `TreeItem` of the cell. You can set an `onEditCommit` event handler that is fired when the user selects an item. The following snippet of code adds such a handler for the `TreeView` that prints a message on the standard output:

```

// Add an onEditCommit handler
treeView.setOnEditCommit(e -> {

```

```

        System.out.println(e.getTreeItem() + " changed." +
            " old = " + e.getOldValue() +
            ", new = " + e.getNewValue());
    });
}

```

Clicking a selected cell puts the cell into editing mode. Double-clicking an unselected cell puts the cell into editing mode. Changing the focus to another cell or selecting an item from the list puts the editing cell into nonediting mode and the current value is displayed in a Label.

## Editing Data Using a Combo Box

Editing data using a combo box works similar to the method used for a `ChoiceBoxTreeCell`. Please refer to the section “[Editing Data Using a Choice Box](#)” for more details.

A `ComboBoxTreeCell` is rendered as a Label in nonediting mode and as a combo box in editing mode. Its `forTreeView()` static method returns a cell factory. The method is overloaded. You need to pass the list of items to be shown in the combo box. If the `toString()` method of the item does not return a user-friendly string for the user, use a string converter. The following snippet of code sets a cell factory for a `TreeView` that will use instances of `ComboBoxTreeCell` to render `TreeItems`. In editing mode, the combo box will display three items: Item-1, Item-2, and Item-3.

```

TreeView<String> treeView = new TreeView<>();
treeView.setCellFactory(ComboBoxTreeCell.<String>forTreeView("Item-1", "Item-2", "Item-3"));

```

## Editing Data Using a *TextField*

A `TextFieldTreeCell` is rendered as a Label in nonediting mode and as a `TextField` in editing mode. Its `forTreeView()` static method returns a cell factory. The method is overloaded. Use a string converter if the item type is not `String`. The following snippet of code sets a cell factory for a `TreeView` that will use instances of `TextFieldTreeCell` to render `TreeItems`. In editing mode, the `TextField` will display the item value.

```

TreeView<String> treeView = new TreeView<>();
treeView.setCellFactory(TextFieldTreeCell.forTreeView());

```

Clicking a selected cell or double-clicking an unselected cell puts the cell into editing mode, which displays the cell data in a `TextField`. Once the cell is in editing mode, you need to click in the `TextField` (one more click!) to put the caret in the `TextField` so you can make changes.

**Tip** Double-clicking a cell representing a branch node will not put the cell in editing mode. Rather, the cell is expanded or collapsed. The trick is to use two

single clicks instead of a double-click on a branch node. Both a double-click and two single clicks put a cell of a leaf node in editing mode.

If you are in the middle of editing a cell data, press the Esc key to cancel editing, which will return the cell to nonediting mode and reverts to the old data in the cell. Pressing the Enter key commits the data to the TreeItem for the cell.

If you are editing a cell using a TextFieldTreeCell, moving the focus to another cell, for example, by clicking another cell, cancels the editing and puts the old value back in the cell. This is not what a user expects. At present, there is no easy solution to this problem. You will have to create a subclass of TreeCell and add a focus change listener so you can commit the data when the TextField loses focus.

The program in Listing 14-6 shows how to use TextFieldTreeCell to edit cell data in a TreeView. Run the application and click a cell two times to put the cell in editing mode. A TextField will display the cell data. Change the data and press the Enter key to commit the changes. The program adds edit-related event handlers to the TreeView that prints a message on the standard output when the events occur. Figure 14-4 shows the cell data being edited in a TextFieldTreeCell.

#### ***Listing 14-6.*** Using TextFieldTreeCell to Edit Cell Data in a TreeView

```
// TreeViewEditingData.java
package com.jdojo.control;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.TreeView;
import javafx.scene.control.cell.TextFieldTreeCell;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class TreeViewEditingData extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        TreeView<String> treeView
= TreeViewUtil.getTreeView();

        // Make the TreeView editable
        treeView.setEditable(true);
    }
}
```

```
// Set a cell factory to use TextFieldTreeCell
treeView.setCellFactory(TextFieldTreeCell.forTreeView
());

// Set editing related event handlers
treeView.setOnEditStart(this::editStart);
treeView.setOnEditCommit(this::editCommit);
treeView.setOnEditCancel(this::editCancel);

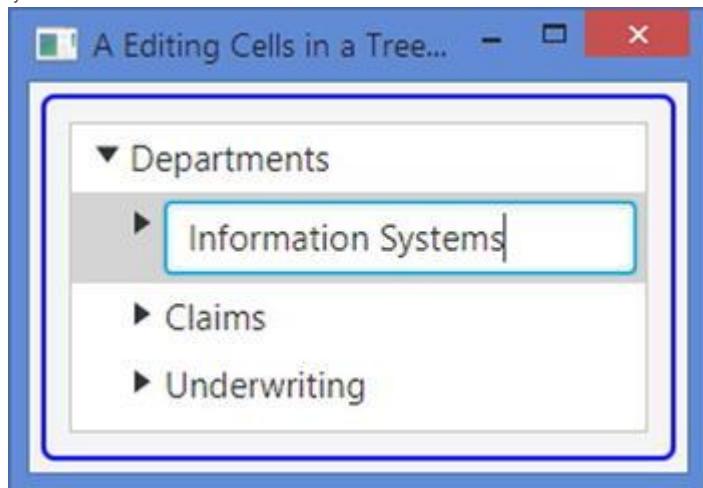
HBox root = new HBox(treeView);
root.setStyle("-fx-padding: 10;" +
    "-fx-border-style: solid inside;" +
    "-fx-border-width: 2;" +
    "-fx-border-insets: 5;" +
    "-fx-border-radius: 5;" +
    "-fx-border-color: blue;");

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("A Editing Cells in a TreeView");
stage.show();
}

public void editStart(TreeView.EditEvent<String> e) {
    System.out.println("Started editng: "
+ e.getTreeItem() );
}

public void editCommit(TreeView.EditEvent<String> e) {
    System.out.println(e.getTreeItem() + " changed." +
        " old = " + e.getOldValue() +
        ", new = " + e.getNewValue());
}

public void editCancel(TreeView.EditEvent<String> e) {
    System.out.println("Cancelled editng: "
+ e.getTreeItem() );
}
```



**Figure 14-4.** A cell data being edited in a `TextFieldTreeCell`

## Loading `TreeItems` on Demand

So far in the examples, you have been loading all items in a `TreeView` at once. Sometimes the number of items is too big or unknown. In those cases, you would need to load the items when the user expands a node to make efficient use of memory. However, this approach is a little complex to implement. In this section, you will develop a file system browser that will create nodes on demand.

First, you need to create a class inheriting from the `TreeItem` class. Listing 14-7 has the code for this new class `PathTreeItem`, which inherits from `TreeItem<Path>`. The class needs to override the `getChildren()` and `isLeaf()` methods of the `TreeItem` class. The three instance variables are used to cache the results of the `isLeaf()` method call and flags indicating that the methods were called once. The constructor calls the constructor of the `TreeItem` class and sets an icon for the node depending on whether it is a file or a directory. The `populateChildren()` method contains the main logic for populating a node. The root directories for the default file system are added as children for the root node. A nonroot node is populated with its subdirectories and subfiles.

**Note** This program will not refresh the items if they change in the file system after it loads them because you load children for a node only once. This task is left to you. As an exercise, you will need to modify the `PathTreeItem` class to implement the refresh functionality using the watch-service for root directories, which was introduced in Java 7. A trivial, inefficient implementation would be to load children every time the `getChildren()` method is called.

### **Listing 14-7.** The `PathTreeItem` Class, as an Implementation of the `TreeItem<Path>` Class

```
// PathTreeItem.java
package com.jdojo.control;

import java.io.IOException;
import java.nio.file.FileSystems;
import java.nio.file.Files;
import java.nio.file.Path;
import javafx.collections.ObservableList;
import javafx.scene.control.TreeItem;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;

public class PathTreeItem extends TreeItem<Path>{
```

```

private boolean childrenLoaded = false;
private boolean leafPropertyComputed = false;
private boolean leafNode = false;

public PathTreeItem(Path path) {
    super(path);
    ImageView icon = null;
    if (Files.isDirectory(path)) {
        icon = getFolderIcon("folder.jpg");
    } else {
        icon = getFolderIcon("file.jpg");
    }
    this.setGraphic(icon);
}

@Override
public ObservableList<TreeItem<Path>> getChildren() {
    if (!childrenLoaded) {
        childrenLoaded = true;
        populateChildren(this);
    }
    return super.getChildren();
}

@Override
public boolean isLeaf() {
    if (!leafPropertyComputed) {
        leafPropertyComputed = true;
        Path path = this.getValue();
        leafNode = !Files.isDirectory(path);
    }
    return leafNode;
}

private void populateChildren(TreeItem<Path> item) {
    item.getChildren().clear();
    if (item.getParent() == null) {
        // Add root directories
        for (Path p :
FileSystems.getDefault().getRootDirectories()) {
            item.getChildren().add(new
PathTreeItem(p));
        }
    } else {
        Path path = item.getValue();
        // Populate sub-directories and files
        if (Files.isDirectory(path)) {
            try {
                Files.list(path).forEach(
                    p ->
item.getChildren().add(new PathTreeItem(p)));
            }
            catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

```

```

        }
    }
}

private ImageView getFolderIcon(String fileName) {
    ImageView imgView = null;
    try {
        String imagePath = "resources/picture/"
+ fileName;
        Image img = new Image(imagePath);
        imgView = new ImageView(img);
    }
    catch (Exception e) {
        e.printStackTrace();
    }
    return imgView;
}
}
}

```

Once you have the `PathTreeItem` class, building a file system browser is easy. The program in Listing 14-8 creates a `TreeView` that lets you browse the default file system on your machine. It creates a root node with the current directory. You can create the root node for any directory, because you do not show the root node. The following snippet of code creates a `TreeView` with the current directory as its root node:

```
PathTreeItem rootNode = new PathTreeItem(Paths.get("."));
TreeView<Path> treeView = new TreeView<>(rootNode);
```

You would then hide the root node, so the user can start browsing the file system:

```
treeView.setShowRoot(false);
```

Then set a cell factory that displays only the name of the file instead of its path. If you want to see the path of all the files, you may comment the statement setting the cell factory:

```
// Set a cell factory to display only file name
treeView.setCellFactory(...);
```

### ***Listing 14-8.*** A File System Browser

```
// FileSystemBrowser.java
package com.jdojo.control;

import java.nio.file.Path;
import java.nio.file.Paths;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.TreeCell;
import javafx.scene.control.TreeView;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;
```

```

public class FileSystemBrowser extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // Create a root node using the current directory.
        PathTreeItem rootNode = new
PathTreeItem(Paths.get("."));
        TreeView<Path> treeView = new TreeView<>(rootNode);
        treeView.setShowRoot(false);

        // Set a cell factory to display only file name
        treeView.setCellFactory((TreeView<Path> tv) -> {
            TreeCell<Path> cell = new TreeCell<Path>() {
                @Override
                public void updateItem(Path item, boolean
empty) {
                    super.updateItem(item, empty);
                    if (item != null && !empty) {
                        Path fileName
= item.getFileName();
                        if (fileName == null) {
                            this.setText(ite
m.toString());
                        } else {
                            this.setText(fil
eName.toString());
                        }
                        this.setGraphic(
this.getTreeItem().getGra
phic());
                    } else {
                        this.setText(null);
                        this.setGraphic(null);
                    }
                }
            };
            return cell;
        });
    }

    HBox root = new HBox(treeView);
    root.setStyle("-fx-padding: 10;" +
"-fx-border-style: solid inside;" +
"-fx-border-width: 2;" +
"-fx-border-insets: 5;" +
"-fx-border-radius: 5;" +
"-fx-border-color: blue;");
}

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("File System Browser");
stage.show();

```

```

    }
}

```

## Scrolling to a *TreeItem*

`TreeView` automatically provides vertical and horizontal scrollbars when needed. Users can use the scrollbars to scroll to a specific item. Sometimes you may need programmatic support for scrolling. For example, when you add a `TreeItem` to a `TreeView`, you may want the `TreeItem` visible to the user by scrolling it to the view. Use the `scrollTo(int rowIndex)` method of the `TreeView` class to scroll the `TreeItem` at the specified `rowIndex` to the view. You can get the row index of a `TreeItem` using the `getRow(TreeItem<T> item)` method.

`TreeView` fires a `ScrollToEvent` when there is a request to scroll to a row index using the `scrollTo()` method.

The `ScrollToEvent` class contains a `getScrollTarget()` method that returns the row index that was passed to the `scrollTo()` method.

**Tip** The `ScrollToEvent` is not fired when the user scrolls using the vertical scrollbar. It is fired when the `scrollTo()` method is used to scroll.

The following snippet of code sets a `ScrollToEvent` handler for a `TreeView` that prints the `TreeItem` and its row index to which the scrolling was requested:

```

TreeView<String> treeView = new TreeView<String>();
...
treeView.setOnScrollTo(e -> {
    int rowIndex = e.getScrollTarget();
    TreeItem<String> item = treeView.getTreeItem(rowIndex);
    System.out.println("Scrolled to: " + item.getValue() + at "
+ rowIndex);
});

```

## *TreeView* Selection Model

`TreeView` uses a selection model to select one or multiple `TreeItems`. The `selectionModel` property represents the selection model. The default selection model is an instance of the abstract class `MultipleSelectionModel`. The following snippet of code enables multiple selection. Press the Ctrl or Shift key while clicking a node to select multiple nodes.

```

TreeView<String> treeView = new TreeView<>();
// Enable mutiple selection for the TreeView
treeView.getSelectionModel().setSelectionMode(SelectionMode.MULTI
PLE);

```

Please refer to the API documentation of the `MultipleSelectionModel` class for more details on how to select `TreeItems` and how to get the selected `TreeItems`.

## **Styling `TreeView` with CSS**

The default CSS style-class name for a `TreeView` is `tree-view`. `TreeView` does not add any CSS pseudo-classes or properties. It inherits them from the `Control`.

A `TreeView` uses instances of `TreeCell` to display the `TreeItems`. Mostly you style the `TreeCells` in the `TreeView`. The default CSS style-class for `TreeCell` is `tree-cell`.

`TreeCell` contains an `-fx-indent` property, which is the amount of space to multiply by the level of the cell to get its left margin. The default value is `10px`.

`TreeCell` supports two CSS pseudo-classes:

- `expanded`
- `collapsed`

The `expanded` pseudo-class is applied when the cell is expanded. The `collapsed` pseudo-class applies when the cell is not expanded.

The following style sets the text color to blue and font size to `10pt` for a `TreeCell`:

```
.tree-cell {
    -fx-text-fill: blue;
    -fx-font-size: 10pt;
}
```

The style-class name for the disclosure node in a cell is `tree-disclosure-node`. It has a substructure named `arrow`, which is the triangle showing the expanded state of the node. You can change the triangle to a plus or minus sign icon using the following styles. The code assumes that the image files are in the same directory as the CSS file containing the styles:

```
.tree-cell .tree-disclosure-node .arrow {
    -fx-shape: null;
    -fx-background-color: null;
    -fx-background-image: url("plus_sign.jpg");
}

.tree-cell:expanded .tree-disclosure-node .arrow {
    -fx-shape: null;
    -fx-background-color: null;
    -fx-background-image: url("minus_sign.jpg");
}
```

You can also set the shape of the disclosure node in CSS using the SVG path. The following code sets plus and minus signs as the disclosure nodes for expanded and collapsed nodes, respectively. Figure 14-5 shows a TreeView using these styles.

```
.tree-cell .tree-disclosure-node .arrow {
    -fx-shape: "M0 -0.5 h2 v2 h1 v-2 h2 v-1 h-2 v-2 h-1 v2 h-2
v1z";
}

.tree-cell:expanded .tree-disclosure-node .arrow {
    -fx-shape: "M0 -0.5 h5 v-1 h-5 v1z";
    -fx-padding: 4 0.25 4 0.25;
}
```

## **- Departments**

**+ IS**

**- Claims**

**Lael Boyd**

**Janet Biddle**

**+ Underwriting**

**Figure 14-5.** A TreeView using plus and minus signs for expanded and collapsed disclosure nodes

## **Summary**

A TreeView is a control that displays hierarchical data in a tree-like structure. You can think of a TreeView as displaying a tree upside down—the root of the tree being at the top. Each item in a TreeView is an instance of the TreeItem class. TreeItems form parent-child relationships. A TreeItem is also referred to as a node.

The TreeItem class does not inherit from the Node class. Therefore, a TreeItem is not a JavaFX Node and it cannot be added to a scene graph. A TreeItem is categorized as a branch or leaf node. If a TreeItem contains other instances of TreeItem, which are called its children, it is called a branch node. Otherwise, it is called a leaf node. A branch node can be in an expanded or collapsed state.

A TreeItem serves as the data model in a TreeView. Each TreeItem uses an instance of the TreeCell class to render its

value. TreeCells in a TreeView can be customized using a cell factory. By default, a TreeCell is not editable.

TreeView is a virtualized control. It creates only as many instances of TreeCell as needed to display the items for its current height. Cells are recycled as you scroll through items. Virtualization makes it possible to use TreeView for viewing very large number of items without using a large amount of memory. Note, however, that loading TreeItems always takes memory. Virtualization helps only in viewing the items by recycling the cells used in viewing them.

The first item in a TreeView that does not have a parent is known as the root node. By default, the root node is visible. Calling the `setShowRoot (false)` method of the TreeView hides the root node. Hiding the root node makes traversing the TreeView a little easier because the user has one less level of indentation to traverse. Hiding the root node shows its child nodes at the first level.

A TreeItem fires events as it is modified, for example, by adding or removing children or expanding or collapsing. An instance of the `TreeModificationEvent` class, which is a static inner class of the `TreeItem` class, represents all kinds of modification events.

Adding and removing TreeItems is as easy as adding or removing them in the children list of their parents. The root node does not have a parent. To delete the root node, you need to set the `root` property of the TreeView to null.

TreeView uses a TreeCell to render a TreeItem. A TreeCell is an `IndexedCell`. You can visualize items in a TreeView from top to bottom arranged in rows. Each row has exactly one item. Each item is given a row index. The first item, which is the root item, has an index of zero. The row indices are given only to the visible items. TreeView contains a read-only `expandedItemCount` property that is the number of visible items. Use the `getExpandedItemCount ()` method to get the number of visible items. If a node above an item is expanded or collapsed, the index of the item changes to reflect new visible items. The index of a TreeCell in a TreeView and the row index of an item are the same. Use the `getIndex ()` method of the TreeCell or the `getRow (TreeItem<T> item)` method of the TreeView to get the row index of an item. A TreeCell is a Labeled control. By default, it uses the following rules to render its TreeItem: If the value in the TreeItem is an instance of the `Node` class, the value is displayed using the `graphic` property of the cell. Otherwise,

the `toString()` method of the value is called and the returned string is displayed using the `text` property of the cell.

A cell in a `TreeView` can be editable. An editable cell may switch between editing and nonediting mode. In editing mode, cell data can be modified by the user. For a cell to enter editing mode, the `TreeView` must be editable. `TreeView` has an `editable` property, which can be set to true using the `setEditable(true)` method. By default, `TreeView` is not editable. Creating a `TreeView` does not let you edit its cells. Cell-editing capability is provided through specialized implementations of the `TreeCell` class. The JavaFX library provides some of these implementations. Set the cell factory for a `TreeView`, which is a `Callback` object, to use one of the following implementations of the `TreeCell` to make cells in a `TreeView` editable: `CheckBoxTreeCell`, `ChoiceBoxTreeCell`, `ComboBoxTreeCell`, or `TextFieldTreeCell`.

`TreeView` lets you load all items at once or items on demand. `TreeView` automatically provides vertical and horizontal scrollbars when needed. `TreeView` uses a selection model to select one or multiple `TreeItems`. The `selectionModel` property represents the selection model. `TreeView` supports styling using CSS.

The next chapter will discuss the control called `TreeTableView`.

## CHAPTER 15



### Understanding *TreeTableView*

In this chapter, you will learn:

- What a *TreeTableView* is
- How to set up the model for a *TreeTableView*
- How to create a *TreeTableView*, add columns to it, and populate it with data
- How to sort data in a *TreeTableView*
- How to show and hide columns in a *TreeTableView*
- How to customize cells in a *TreeTableView*
- How to use the selection model of a *TreeTableView*
- How to edit data and add or delete rows in a *TreeTableView*

If you are not already familiar with *TableView* and *TreeView* controls, I suggest you review that before proceeding with this chapter.

### What Is a *TreeTableView*?

The *TreeTableView* control combines the features of the *TableView* and *TreeView* controls. It displays a *TreeView* inside a *TableView*. A *TreeView* is used to view hierarchical data; a *TableView* is used to view tabular data. A *TreeTableView* is used to view hierarchical data in a tabular form, as shown in Figure 15-1.

First Name	Last Name	Birth Date
▼ Ram	Singh	1930-01-01
▶ Janki	Sharan	1956-12-17
▶ Sita	Sharan	1961-03-01
Kishori	Sharan	1968-01-12
Ratna	Sharan	1978-04-14

**Figure 15-1.** A TreeTableView showing family hierarchy with details

TreeTableView inherits from Control, not from TreeView or TableView. TreeTableView reuses most of the code used for TreeView and TableView. Most of the classes in the API are inherited from a common abstract base class for all three controls. For example, the TableColumn and TreeTableColumn classes are used to define columns in TableView and TreeTableView, respectively, and both are inherited from the TableColumnBase class.

TreeTableView API looks huge as it combines the APIs for both TreeView and TableView. However, if you are familiar with TreeView and TableView APIs, the TreeTableView API will look familiar to you. I will not discuss all features of TreeTableView, as they will be a repetition of what I have already discussed for TreeView and TableView. TreeTableView supports the following features:

- You can add multiple columns.
- You can have nested columns.
- You can resize columns at runtime.
- You can reorder columns at runtime.
- You can sort data on a single or multiple columns.
- You can add a context menu for columns.
- You can set a *cell value factory* for a column to populate its cells.
- You can set a *cell factory* for a column to customize its cells rendering.

- You can edit data in cells.

## Model for *TreeTableView*

TreeItems provide the model in a TreeView. Each node in the TreeView derives its data from the corresponding TreeItem. Recall that you can visualize each node (or TreeItem) in a TreeView as a row with only one column.

An ObservableList provides the model in a TableView. Each item in the observable list provides data for a row in the TableView. A TableView can have multiple columns.

TreeTableView also uses a model for its data. Because it is a combination of TreeView and TableView, it has to decide which type of model it uses. It uses the model based on TreeView. That is, each row in a TreeTableView is defined by a TreeItem in a TreeView. TreeTableView supports multiple columns. Data for columns in a row are derived from the TreeItem for that row. Table 15-1 compares the model support for the three controls.

**Table 15-1.** Comparing the Model Support for TreeView, TableView, and TreeTableView

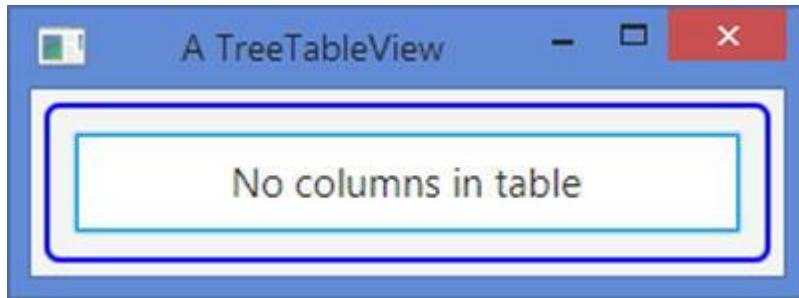
	<b>TreeView</b>	<b>TableView</b>	<b>TreeTableView</b>
<b>Model</b>	TreeItems	An ObservableList	TreeItems
<b>Row</b>	A TreeItem	An item from the ObservableList	A TreeItem
<b>Column</b>	Only one column	Multiple columns	Multiple columns

## Creating a *TreeTableView*

An instance of the TreeTableView represents a TreeTableView control. The class takes a generic type argument, which is the type of the item contained in the TreeItems. Recall that TreeItems provide a model for a TreeTableView. The generic type of the controls and its TreeItems are the same.

The TreeTableView class provides two constructors. The default constructor creates a TreeTableView with no data. The following statement creates a TreeTableView of Person, which is shown in Figure 15-2. The control displays a placeholder, similar to the one shown by TableView. Like a TableView, TreeTableView contains a placeholder property, which is Node, and if you need to, you can supply your own placeholder:

```
// Create a TableView
TreeTableView<Person> treeTable = new TreeTableView<>();
```



**Figure 15-2.** A `TreeTableView` without a column and data

An instance of the `TreeTableColumn` class represents a column in a `TreeTableView`. The `getColumns()` method of the `TreeTableView` class returns an `ObservableList` of `TreeTableColumns`, which are columns that are added to the `TreeTableView`. You need to add columns to this columns list. The following snippet of code creates three columns and adds them to the `TreeTableView`. The resulting `TreeTableView` is shown in Figure 15-3, which shows a placeholder stating that the content (or data/model) is missing.

```
// Create three columns
TreeTableColumn<Person, String> fNameCol = new
TreeTableColumn<>("First Name");
TreeTableColumn<Person, String> lNameCol = new
TreeTableColumn<>("Last Name");
TreeTableColumn<Person, String> bDateCol = new
TreeTableColumn<>("Birth Date");

// Add columns to the TreeTableView
treeTable.getColumns().addAll(fNameCol, lNameCol, bDateCol);
```



**Figure 15-3.** A `TreeTableView` with three columns, but no content

Now you need to supply data for the control. `TreeTableView` displays hierarchical data in tabular form. It requires you to construct a hierarchical model using `TreeItems`. You

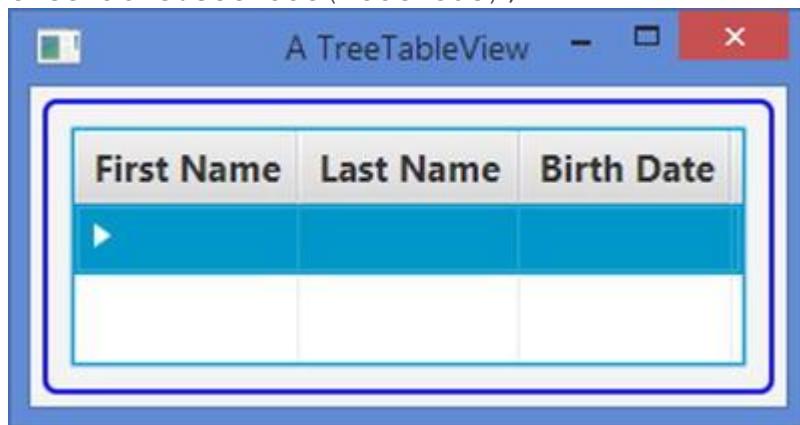
need to pass the root TreeItem to the TreeTableView. Like a TreeView, a TreeTableView contains a root property, which is the root TreeItem for the TreeView. The root property acts as a model for the TreeTableView to supply it data.

The following snippet of code creates a tree of some persons. The root TreeItem is set as the root of the TreeTableView. The resulting TreeTableView is shown in Figure 15-4.

```
// Create TreeItems
Person ram = new Person("Ram", "Singh", LocalDate.of(1930, 1,
1));
Person janki = new Person("Janki", "Sharan", LocalDate.of(1956,
12, 17));
Person sita = new Person("Sita", "Sharan", LocalDate.of(1961, 3,
1));
TreeItem<Person> rootNode = new TreeItem<>(ram);
TreeItem<Person> jankiNode = new TreeItem<>(janki);
TreeItem<Person> sitaNode = new TreeItem<>(sita);

// Add children to the root node
rootNode.getChildren().addAll(jankiNode, sitaNode);

// Set the model for the TreeTableView
treeTable.setRoot(rootNode);
```



**Figure 15-4.** A TreeTableView with columns, root node, but without cell value factory

You have made progress! The placeholder has disappeared and you now see a disclosure node (a triangle) in the first column. However, you still do not see any data. You have columns and model (TreeItems). There is a missing link and the columns do not know how to extract data from the TreeItems. This is accomplished by setting the *cell value factory* for each column. Setting the cell value factory for a TreeTableColumn is very similar to of the way you would for TableColumn. The following snippet of code sets the cell value factory for columns:

```
// Set the cell value factory for columns
fNameCol.setCellValueFactory(new
```

```
TreeItemPropertyValueFactory<>("firstName")) ;
lNameCol.setCellValueFactory(new
TreeItemPropertyValueFactory<>("lastName")) ;
bDateCol.setCellValueFactory(new
TreeItemPropertyValueFactory<>("birthDate")) ;
```

A `TreeItemPropertyValueFactory` reads the specified property of the object stored in the `valueProperty` of a `TreeItem` to populate the cells of the column. In the example, each `TreeItem` contains a `Person` object. The resulting `TreeTableView` is shown in Figure 15-5, after you expand the root node in the first column. The `TreeTableView` shows one row corresponding to each expanded `TreeItem`. If you collapse a node, the rows for its children nodes are hidden.

First Name	Last Name	Birth Date
Ram	Singh	1930-01-01
Janki	Sharan	1956-12-17
Sita	Sharan	1961-03-01

**Figure 15-5.** A `TreeTableView` with data

If you ignore the disclosure node and indentations in the first column, this is exactly how a `TableView` shows the data. The disclosure node and the indentations are features of the `TreeView`.

By default, a `TreeTableView` shows the disclosure node in the first column. You can show it in any other column using the `treeColumn` property. The following snippet of code shows the disclosure node in the Last Name column. The following snippet sets the `treeColumn` property to `lNameCol`, so the disclosure node is shown in the Last Name column, as shown in Figure 15-6.

```
// Show the disclosure node in the Last Name column
treeTable.setTreeColumn(lNameCol);
```

First Name	Last Name	Birth Date
Ram	▼ Singh	1930-01-01
Janki	Sharan	1956-12-17
Sita	Sharan	1961-03-01

**Figure 15-6.** A column other than the first column showing the disclosure node

Another constructor of the `TreeTableView` class takes the value for its `root` property as an argument. You can use it as follows:

```
TreeTableView<Person> treeTable = new
TreeTableView<Person>(rootNode);
```

You will be using data for a family tree as the model for most of the examples in this chapter. Let's use the reusable code for creating the mode and columns in a `TreeTableUtil` class as shown in Listing 15-1. The class consists of all static methods. The `getModel()` method constructs the family tree and returns the root node of the tree. All other methods create a column, set the cell value factory for the column, and return the column reference.

### **Listing 15-1.** A Utility Class to Supply Model and Columns for a `TreeTableView` of Persons

```
// TreeTableUtil.java
package com.jdojo.control;

import com.jdojo.mvc.model.Person;
import java.time.LocalDate;
import javafx.scene.control.TreeTableColumn;
import javafx.scene.control.TreeItem;
import javafx.scene.control.cell.TreeItemPropertyValueFactory;

public class TreeTableUtil {
    /* Returns a root TreeItem for a family members */
    @SuppressWarnings("unchecked")
    public static TreeItem<Person> getModel() {
        /* Create all persons */
        // First level
        Person ram = new Person("Ram", "Singh",
LocalDate.of(1930, 1, 1));

        // Second level
```

```

        Person janki = new Person("Janki", "Sharan",
LocalDate.of(1956, 12, 17));
        Person sita = new Person("Sita", "Sharan",
LocalDate.of(1961, 3, 1));
        Person kishori = new Person("Kishori", "Sharan",
LocalDate.of(1968, 1, 12));
        Person ratna = new Person("Ratna", "Sharan",
LocalDate.of(1978, 4, 14));

        // Third level
        Person navin = new Person("Navin", "Sharan",
LocalDate.of(1980, 5, 10));
        Person vandana = new Person("Vandana", "Sharan",
LocalDate.of(1981, 3, 20));
        Person neeraj = new Person("Neeraj", "Sharan",
LocalDate.of(1982, 6, 3));

        Person gaurav = new Person("Gaurav", "Sharan",
LocalDate.of(1990, 8, 27));
        Person saurav = new Person("Saurav", "Sharan",
LocalDate.of(1994, 5, 15));

        // Fourth level
        Person palak = new Person("Palak", "Sharan",
LocalDate.of(2010, 6, 3));
        Person ashwin = new Person("Ashwin", "Sharan",
LocalDate.of(2012, 10, 11));
        Person advik = new Person("Advik", "Sharan",
LocalDate.of(2012, 10, 11));

        // Build nodes
        TreeItem<Person> navinNode = new TreeItem<>(navin);
        navinNode.getChildren().addAll(new
TreeItem<>(ashwin), new TreeItem<>(advik));
        TreeItem<Person> vandanaNode = new
TreeItem<>(vandana);
        vandanaNode.getChildren().addAll(new
TreeItem<>(palak));

        TreeItem<Person> jankiNode = new TreeItem<>(janki);
        jankiNode.getChildren().addAll(navinNode, new
TreeItem<>(neeraj), vandanaNode);

        TreeItem<Person> sitaNode = new TreeItem<>(sita);
        sitaNode.getChildren().addAll(new
TreeItem<>(gaurav), new TreeItem<>(saurav));

        TreeItem<Person> kishoriNode = new
TreeItem<>(kishori);
        TreeItem<Person> ratnaNode = new TreeItem<>(ratna);

        // Create the root node and add children
        TreeItem<Person> rootNode = new TreeItem<>(ram);
        rootNode.getChildren().addAll(jankiNode, sitaNode,
kishoriNode, ratnaNode);
    
```

```

        return rootNode;
    }

    /* Returns Person Id TreeTableColumn */
    public static TreeTableColumn<Person, Integer>
getIdColumn() {
    TreeTableColumn<Person, Integer> idCol = new
TreeTableColumn<>("Id");
    idCol.setCellValueFactory(new
TreeItemPropertyValueFactory<>("personId"));
    return idCol;
}

/* Returns First Name TreeTableColumn */
public static TreeTableColumn<Person, String>
getFirstNameColumn() {
    TreeTableColumn<Person, String> fNameCol = new
TreeTableColumn<>("First Name");
    fNameCol.setCellValueFactory(new
TreeItemPropertyValueFactory<>("firstName"));
    return fNameCol;
}

/* Returns Last Name TreeTableColumn */
public static TreeTableColumn<Person, String>
getLastNameColumn() {
    TreeTableColumn<Person, String> lNameCol = new
TreeTableColumn<>("Last Name");
    lNameCol.setCellValueFactory(new
TreeItemPropertyValueFactory<>("lastName"));
    return lNameCol;
}

/* Returns Birth Date TreeTableColumn */
public static TreeTableColumn<Person, LocalDate>
getBirthDateColumn() {
    TreeTableColumn<Person, LocalDate> bDateCol =
        new TreeTableColumn<>("Birth Date");
    bDateCol.setCellValueFactory(new
TreeItemPropertyValueFactory<>("birthDate"));
    return bDateCol;
}

/* Returns Age Category TreeTableColumn */
public static TreeTableColumn<Person, Person.AgeCategory>
getAgeCategoryColumn() {
    TreeTableColumn<Person, Person.AgeCategory> bDateCol =
        new TreeTableColumn<>("Age Category");
    bDateCol.setCellValueFactory(new
TreeItemPropertyValueFactory<>("ageCategory"));
    return bDateCol;
}
}

```

Listing 15-2 contains a complete program that shows how to create a TreeTableView. It uses the TreeTableUtil class to get the model and columns. Run the program and play with sorting, reordering, and resizing of the columns. Running the program results in a window as shown in Figure 15-7.

**Listing 15-2.** Using a TreeTableView

```
// TreeTableViewTest.java
package com.jdojo.control;

import com.jdojo.mvc.model.Person;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.TreeItem;
import javafx.scene.control.TreeTableView;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class TreeTableViewTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    @SuppressWarnings("unchecked")
    public void start(Stage stage) {
        TreeItem<Person> rootNode
= TreeTableUtil.getModel();
        rootNode.setExpanded(true);

        // Create a TreeTableView with model
        TreeTableView<Person> treeTable = new
TreeTableView<>(rootNode);
        treeTable.setPrefWidth(400);

        // Add columns
        treeTable.getColumns().addAll(TreeTableUtil.getFirst
NameColumn(),
                                         TreeTableUtil.getLastnameColu
mn(),
                                         TreeTableUtil.getBirthdateCol
umn(),
                                         TreeTableUtil.getAgeCategoryC
olumn());}

        HBox root = new HBox(treeTable);
        root.setStyle("-fx-padding: 10;" +
                     "-fx-border-style: solid inside;" +
                     "-fx-border-width: 2;" +
                     "-fx-border-insets: 5;" +
                     "-fx-border-radius: 5;" +
                     "-fx-border-color: blue;");
```

```

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using a TreeTableView");
        stage.show();
    }
}

```

First Name	Last Name	Birth Date	Age Category
▼ Ram	Singh	1930-01-01	SENIOR
▶ Janki	Sharan	1956-12-17	SENIOR
▶ Sita	Sharan	1961-03-01	SENIOR
Kishori	Sharan	1968-01-12	ADULT
Ratna	Sharan	1978-04-14	ADULT

**Figure 15-7.** A *TreeTableView* with its root node expanded

## Sorting Data in a *TreeTableView*

*TreeTableView* supports sorting the same way *TableView* supports sorting. Please refer to the sorting section of Chapter 14 for more in-depth discussion on sorting data. Note that the model for a *TreeTableView* is hierarchical. The hierarchy is always maintained whether or not the model is sorted. For example, the root node is always sorted at the top, irrespective of the sorting criteria used. Sorting in a *TreeTableView* is applied only to the immediate children of each branch node, thus maintaining the hierarchy.

## Populating a *TreeTableColumn* with Data

The `cellValueFactory` property of the *TreeTableColumn* is responsible for populating cells in the column. It is a `Callback` object. The `call()` method receives an object of the `CellDataFeatures` class, which is an inner static class of the *TreeTableColumn* class, and returns an `ObservableValue`. The `getValue()` method of the `CellDataFeatures` class returns the reference of the *TreeItem* for the row. The following snippet of code creates an `Age`

column and sets its cell value factory to compute the age in years of a Person using the birth date:

```
TreeTableColumn<Person, String> ageCol = new TreeTableColumn<>("Age");
ageCol.setCellValueFactory(cellData -> {
    Person p = cellData.getValue().getValue();
    LocalDate dob = p.getBirthDate();
    String ageInYear = "Unknown";
    if (dob != null) {
        long years = YEARS.between(dob, LocalDate.now());
        if (years == 0) {
            ageInYear = "< 1 year";
        } else if (years == 1) {
            ageInYear = years + " year";
        } else {
            ageInYear = years + " years";
        }
    }
    return new ReadOnlyStringWrapper(ageInYear);
});
```

In Listing 15-1, you learned how to use

a `TreeItemPropertyValueFactory` object to set a cell value factory if the values for a column come directly from a property of the `TreeItem` value. For an in-depth discussion of setting the cell value factory, please refer to Chapter 13, which discusses the `TableView` control.

## Showing and Hiding Columns

Showing and hiding columns in a `TreeTableView` work the same way they do for `TableView`. By default, all columns in a `TreeTableView` are visible. The `TreeTableColumn` class has a `visible` property to set the visibility of a column. If you turn off the visibility of a parent column, a column with nested columns, all its nested columns will become invisible. The following code shows this:

```
TreeTableColumn<Person, String> idCol = new TreeTableColumn<>("Id");

// Make the Id column invisible
idCol.setVisible(false);
...
// Make the Id column visible
idCol.setVisible(true);
```

Sometimes you may want to let the user control the visibility of columns. The `TreeTableView` class has a `tableMenuButtonVisible` property. If it is set to true, a menu button is displayed in the header area. Clicking the Menu button displays

a list of all leaf columns. Columns are displayed as radio menu items that can be used to toggle their visibility.

## Customizing Data Rendering in Cells

A cell in a `Tree TableColumn` is an instance of the `Tree TableCell` class, which displays the data in the cell. A `Tree TableCell` is a Labeled control, which is capable of displaying text, a graphic, or both.

You can specify a cell factory for a `Tree TableColumn`. The job of a cell factory is to render the data in the cell.

The `Tree TableColumn` class contains a `cellFactory` property, which is a `CallbackObject`. Its `call()` method is passed in the reference of the `Tree TableColumn` to which the cell belongs. The method returns an instance of `Tree TableCell`. The `updateItem()` method of the `Tree TableCell` is overridden to provide the custom rendering of the cell data.

`Tree TableColumn` uses a default cell factory if its `cellFactory` property is not specified. The default cell factory displays the cell data depending on the type of the data. If the cell data is a `Node`, the data are displayed in the `graphic` property of the cell. Otherwise, the `toString()` method of the cell data is called and the returned string is displayed in the `text` property of the cell.

You can display formatted birth dates in the Birth Date column in the previous example. The Birth Date column is formatted as `yyyy-mm-dd`, which is the default ISO date format returned by the `toString()` method of the `LocalDate` class. You may want to format birth dates in the `mm/dd/yyyy` format. You can achieve this by setting a custom cell factory for the column, as shown in the following code:

```
Tree TableColumn<Person, LocalDate> birthDateCol
= TreeTableUtil.getBirthDateColumn();
birthDateCol.setCellFactory (col -> {
    Tree TableCell<Person, LocalDate> cell = new
    Tree TableCell<Person, LocalDate>() {
        @Override
        public void updateItem(LocalDate item, boolean
empty) {
            super.updateItem(item, empty);
            // Cleanup the cell before populating it
            this.setText(null);
            this.setGraphic(null);
            if (!empty) {
                // Format the birth date in mm/dd/yyyy

```

```

format
        String formattedDob =
        DateTimeFormatter.ofPattern("MM/dd/yyyy")
).format(item);
        this.setText(formattedDob);
    }
}
};

return cell;
});

```

You can use the above technique to display images in cells. In the `updateItem()` method, create an `ImageView` object for the image and display it using the `setGraphic()` method of the `TreeTableCell`.

`TableCell` contains `tableColumn`, `tableRow`, and `tableView` properties that store the references of its  `TableColumn`,  `TableRow`, and  `TableView`, respectively. These properties are useful to access the item in the data model that represents the row for the  `TableCell`.

`TreeTableCell` contains `tableColumn`, `tableRow`, and `treeTableView` properties that store the references of its  `TreeTableColumn`,  `TreeTableRow`, and  `TreeTableView`, respectively. These properties are useful to access the item in the model that represents the row for the cell.

The following subclasses of `TreeTableCell` render cell data in different ways. For example, a  `CheckBoxTreeTableCell` renders cell data in a  `CheckBox` and a  `ProgressBarTreeTableCell` renders a number using a  `ProgressBar`:

- `CheckBoxTreeTableCell`
- `ChoiceBoxTreeTableCell`
- `ComboBoxTreeTableCell`
- `ProgressBarTreeTableCell`
- `TextFieldTreeTableCell`

The  `CheckBox`,  `ChoiceBox`,  `ComboBox`, and  `TextField` versions of the  `XxxTreeTableCell` are used to edit data in cells. I will discuss how to edit data in a  `TreeTableCell` shortly.

The following snippet of code creates a computed column. It sets the cell factory for the column to display a  `CheckBox`. If the Person falls in the baby age category, the  `CheckBox` is selected.

```

// Create a "Baby?" column
TreeTableColumn<Person, Boolean> babyCol = new
TreeTableColumn<>("Baby?");

```

```

babyCol.setCellValueFactory(cellData -> {
    Person p = cellData.getValue().getValue();
    Boolean v = (p.getAgeCategory() ==
Person.AgeCategory.BABY);
    return new ReadOnlyBooleanWrapper(v);
});

// Set a cell factory that will use a CheckBox to render the
value
babyCol.setCellFactory(CheckBoxTreeTableCell.<Person>forTreeTable
Column(babyCol));

```

## Selecting Cells and Rows in a *TreeView*

*TreeView* has a selection model represented by its property called `selectionModel`. A selection model is an instance of the `TreeViewSelectionModel` class, which is an inner static class of the `TreeView` class. The selection model supports cell-level and row-level selection. It also supports two selection modes: single and multiple. In the single selection mode, only one cell or row can be selected at a time. In the multiple-selection mode, multiple cells or rows can be selected. By default, single row selection is enabled. You can enable multirow selection using the following code:

```

TreeView<Person> treeTable = ...

// Turn on multiple-selection mode for the TreeTableView
TreeViewSelectionModel<Person> tsm
= treeTable.getSelectionModel();
tsm.setSelectionMode(SelectionMode.MULTIPLE);

```

The cell-level selection can be enabled by setting the `cellSelectionEnabled` property of the selection model to true, as shown in the following snippet of code. When the property is set to true, the `TreeView` is put in cell-level selection mode and you cannot select an entire row. If multiple-selection mode is enabled, you can still select all cells in a row. However, the row itself is not reported as selected because the `TreeView` is in the cell-level selection mode. By default, cell-level selection mode is false.

```

// Enable cell-level selection
tsm.setCellSelectionEnabled(true);

```

The selection model provides information about the selected cells and rows. The `isSelected(int rowIndex)` method returns true if the row at the specified `rowIndex` is selected. Use the `isSelected(int rowIndex, TableColumn<S, ?> column)` method to determine if a cell at the specified `rowIndex` and `column` is selected.

The `getModelItem(int rowIndex)` method returns the `TreeItem` for the specified `rowIndex`.

The selection model provides several methods to select cells and rows and get the report of selected cells and rows. Please refer to the API documentation for the `TreeTableViewSelectionModel` class for more details.

It is often a requirement to make some changes or take an action when a cell or row selection changes in a `TreeTableView`. For example, a `TreeTableView` may act as a master list in a master-detail data view. When the user selects a row in the master list, you want the data in the detail view to refresh. Several methods of the `TreeTableViewSelectionModel` class return an `ObservableList` of selected indices and items. If you are interested in handling the selection change event, you need to add a `ListChangeListener` to one of those `ObservableLists`. The following snippet of code adds a `ListChangeListener` to the `ObservableList` returned by the `getSelectedIndices()` method to track the row selection change in a `TreeTableView`:

```
TreeTableViewSelectionModel<Person> tsm
= treeTable.getSelectionModel();
ObservableList<Integer> list = tsm.getSelectedIndices();

// Add a ListChangeListener
list.addListener((ListChangeListener.Change<? extends Integer>
change) -> {
    System.out.println("Row selection has changed");
});
```

## Editing Data in a *TableView*

A cell in a `TreeTableView` can be editable. An editable cell switches between editing and nonediting modes. In editing mode, cell data can be modified by the user. In order for a cell to enter editing mode, the `TreeTableView`, `Tree TableColumn`, and `Tree TableCell` must be editable. All three of them have an `editable` property, which can be set to true using the `setEditable(true)` method. By default, `Tree TableColumn` and `Tree TableCell` are editable. To make cells editable in a `TreeTableView`, you need to make the `TreeTableView` editable, as shown in the following code:

```
TreeTableView<Person> treeTable = ...
treeTable.setEditable(true);
```

The `Tree TableColumn` class supports three types of events:

- `onEditStart`
- `onEditCommit`
- `onEditCancel`

The `onEditStart` event is fired when a cell in the column enters editing mode. The `onEditCommit` event is fired when the user successfully commits the editing, for example, by pressing the Enter key in a `TextField`. The `onEditCancel` event is fired when the user cancels the editing, for example, by pressing the Esc key in a `TextField`. The events are represented by an object of the `TreeTableColumn.CellEditEvent` class. The event object encapsulates the old and new values in the cell, the `TreeItem` of the model being edited, `TreeTableColumn`, the `TreeTablePosition` indicating the cell position where the editing is happening, and the reference of the `TreeView`. Use the methods of the `CellEditEvent` class to get these values.

Making a `TreeView` editable does not let you edit its cell data. You need to do a little more of a plumbing job before you can edit data in cells. Cell editing capability is provided through specialized implementations of the `TreeTableCell` class. JavaFX library provides a few of these implementations. Set the cell factory for a column to use one of the following implementations of the `TreeTableCell` to edit cell data:

- `CheckBoxTreeTableCell`
- `ChoiceBoxTreeTableCell`
- `ComboBoxTreeTableCell`
- `TextFieldTreeTableCell`

Now let's look at an example of editing data using a `TextField`. Please refer to the corresponding section for the `TableView` control for an in-depth discussion of editing data using various controls and handling editing related events. The only difference between editing cells in `TableView` and `TreeView` is the cell classes you will need to use: `TableView` uses subclasses of `TableCell` that are named as `XxxTableCell`; `TreeView` uses subclasses of `TreeTableCell` that are named as `XxxTreeTableCell`.

The following snippet of code sets the cell factory for the First Name column to use a `TextField` to edit data in its cells:

```
TreeTableColumn<Person, String> fNameCol  
= TreeTableUtil.getFirstNameColumn();  
fNameCol.setCellFactory(TextFieldTreeTableCell.<Person>forTreeTableColumn());
```

When editing nonstring data in cell, you need to provide a `StringConverter`. The following snippet of code sets a cell factory for a Birth Date column with a `StringConverter`, which converts

a String to a LocalDate and vice versa. The column type is LocalDate. By default, the LocalDateStringConverter assumes a date format of mm/dd/yyyy:

```
TreeTableColumn<Person, LocalDate> birthDateCol
= TreeTableUtil.getBirthDateColumn();
LocalDateStringConverter converter = new
LocalDateStringConverter();
birthDateCol.setCellFactory(
    TextFieldTreeTableCell.<Person,
LocalDate>forTreeTableColumn(converter));
```

The program in Listing 15-3 shows how to make cells in a TreeTableView editable. Run the program and click a cell twice (using two single clicks) to start editing data. Figure 15-8 shows the First Name cell in the third row in edit mode. When you are done editing, press the Enter key to commit the changes or press the Esc key to cancel editing.

### ***Listing 15-3.*** Editing Data in a TreeTableView

```
// TreeTableViewEditing.java
package com.jdojo.control;

import com.jdojo.mvc.model.Person;
import java.time.LocalDate;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.TreeItem;
import javafx.scene.control.TreeTableColumn;
import javafx.scene.control.TreeTableView;
import javafx.scene.control.cell.TextFieldTreeTableCell;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class TreeTableViewEditing extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    @SuppressWarnings("unchecked")
    public void start(Stage stage) {
        // Create the model
        TreeItem<Person> rootNode
= TreeTableUtil.getModel();
        rootNode.setExpanded(true);

        // Create a TreeTableView with a model
        TreeTableView<Person> treeTable = new
TreeTableView<Person>(rootNode);
        treeTable.setPrefWidth(400);
```

```
// Must make the TreeTableView editable
treeTable.setEditable(true);

// Set appropriate cell factories for
TreeTableColumn<Person, String> fNameCol
= TreeTableUtil.getFirstNameColumn();
fNameCol.setCellFactory(TextFieldTreeTableCell.<Person>forTreeTableColumn());

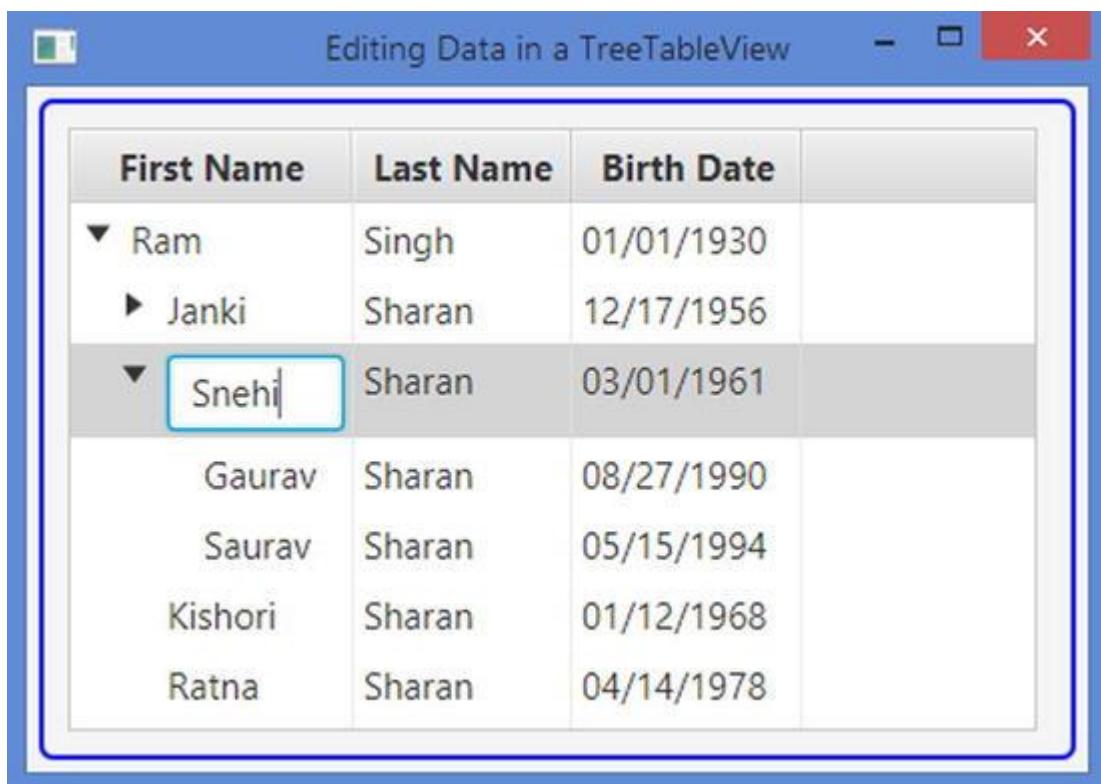
TreeTableColumn<Person, String> lNameCol =
    TreeTableUtil.getLastNameColumn();
lNameCol.setCellFactory(TextFieldTreeTableCell.<Person>forTreeTableColumn());

TreeTableColumn<Person, LocalDate> birthDateCol =
    TreeTableUtil.getBirthDateColumn();
LocalDateStringConverter converter = new
LocalDateStringConverter();
birthDateCol.setCellFactory(
    TextFieldTreeTableCell.<Person,
    LocalDate>forTreeTableColumn(converter));

// Add Columns
treeTable.getColumns().addAll(fNameCol, lNameCol,
birthDateCol);

HBox root = new HBox(treeTable);
root.setStyle("-fx-padding: 10;" +
    "-fx-border-style: solid inside;" +
    "-fx-border-width: 2;" +
    "-fx-border-insets: 5;" +
    "-fx-border-radius: 5;" +
    "-fx-border-color: blue;");

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("Editing Data in a TreeTableView");
stage.show();
}
}
```



**Figure 15-8.** A cell in a `TreeTableView` in edit mode

## Adding and Deleting Rows in a `TableView`

Each row in a `TreeTableView` is represented by a `TreeItem` in its model. Adding and deleting a row in a `TreeTableView` is as simple as adding and deleting `TreeItems` in the model.

The program in Listing 15-4 shows how to add and delete rows. It displays a prebuilt family hierarchy in a `TreeTableView` along with Add and Delete buttons, as shown in Figure 15-9. Clicking the Add button adds a new row as a child row for the selected row. If there is no row, a new root item is added to the tree. The new row is selected, scrolled to the view, and put in editing mode. The `addRow()` method contains the logic for adding a row. The Delete button deletes the selected row. Notice that all child rows of the selected row are deleted.

### **Listing 15-4.** Adding and Deleting Rows in a `TreeTableView`

```
// TreeTableViewAddDeleteRows.java
package com.jdojo.control;

import com.jdojo.mvc.model.Person;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.TreeItem;
```

```

import javafx.scene.control.TreeTableColumn;
import javafx.scene.control.TreeTableView;
import javafx.scene.control.cell.TextFieldTreeTableCell;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import java.time.LocalDate;
import static
javafx.scene.control.TreeTableView.TreeTableViewSelectionModel;
import javafx.scene.control.Label;
import javafx.stage.Stage;

public class TreeTableViewAddDeleteRows extends Application {
    private final TreeTableView<Person> treeTable = new
TreeTableView<>();

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

    @Override
    @SuppressWarnings("unchecked")
    public void start(Stage stage) {
        // Create the model
        TreeItem<Person> rootNode
= TreeTableUtil.getModel();
        rootNode.setExpanded(true);
        treeTable.setRoot(rootNode);
        treeTable.setPrefWidth(400);
        treeTable.setEditable(true);
        treeTable.getSelectionModel().selectFirst();

        // Set appropriate cell factories for columns
        TreeTableColumn<Person, String> fNameCol
= TreeTableUtil.getFirstNameColumn();
        fNameCol.setCellFactory(TextFieldTreeTableCell.<Pers
on>forTreeTableColumn());

        TreeTableColumn<Person, String> lNameCol =
            TreeTableUtil.getLastNameColumn();
        lNameCol.setCellFactory(TextFieldTreeTableCell.<Pers
on>forTreeTableColumn());

        TreeTableColumn<Person, LocalDate> birthDateCol =
            TreeTableUtil.getBirthDateColumn();
        LocalDateStringConverter converter = new
LocalDateStringConverter();
        birthDateCol.setCellFactory(
            TextFieldTreeTableCell.<Person,
LocalDate>forTreeTableColumn(converter));

        // Add Columns
        treeTable.getColumns().addAll(fNameCol, lNameCol,
birthDateCol);

        // Add a placeholder to the TreeTableView.
    }
}

```

```

        // It is displayed when the root node is deleted.
        treeTable.setPlaceholder(new Label("Click the Add
button to add a row."));

        Label msgLbl = new Label("Please select a row to
add/delete.");
        HBox buttons = this.getButtons();
        VBox root = new VBox(msgLbl, buttons, treeTable);
        root.setSpacing(10);
        root.setStyle("-fx-padding: 10;" +
                      "-fx-border-style: solid inside;" +
                      "-fx-border-width: 2;" +
                      "-fx-border-insets: 5;" +
                      "-fx-border-radius: 5;" +
                      "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Adding/Deleting Rows in
a TableView");
        stage.show();
    }

    private HBox getButtons() {
        Button addBtn = new Button("Add");
        addBtn.setOnAction(e -> addRow());

        Button deleteBtn = new Button("Delete");
        deleteBtn.setOnAction(e -> deleteRow());

        return new HBox(20, addBtn, deleteBtn);
    }

    private void addRow() {
        if (treeTable.getExpandedItemCount() == 0) {
            // There is no row in the TableView
            addNewRootItem();
        } else if (treeTable.getSelectionModel().isEmpty())
        {
            System.out.println("Select a row to add.");
            return;
        } else {
            addNewItem();
        }
    }

    private void addNewRootItem() {
        // Add a root Item
        TreeItem<Person> item = new TreeItem<>(new
Person("New", "New", null));
        treeTable.setRoot(item);

        // Edit the item
        this.editItem(item);
    }
}

```

```

private void addNewItem() {
    // Prepare a new TreeItem with a new Person object
    TreeItem<Person> item = new TreeItem<>(new
Person("New", "New", null));

    // Get the selection model
    TreeTableViewSelectionModel<Person> sm
= treeTable.getSelectionModel();

    // Get the selected row index
    int rowIndex = sm.getSelectedIndex();

    // Get the selected TreeItem
    TreeItem<Person> selectedItem
= sm.getModelItem(rowIndex);

    // Add the new item as children to the selected item
    selectedItem.getChildren().add(item);

    // Make sure the new item is visible
    selectedItem.setExpanded(true);

    // Edit the item
    this.editItem(item);
}

private void editItem(TreeItem<Person> item) {
    // Scroll to the new item
    int newIndex = treeTable.getRow(item);
    treeTable.scrollTo(newIndex);

    // Put the first column in editing mode
    TreeTableColumn<Person, ?> firstCol
= treeTable.getColumns().get(0);
    treeTable.getSelectionModel().select(item);
    treeTable.getFocusModel().focus(newIndex,
firstCol);
    treeTable.edit(newIndex, firstCol);
}

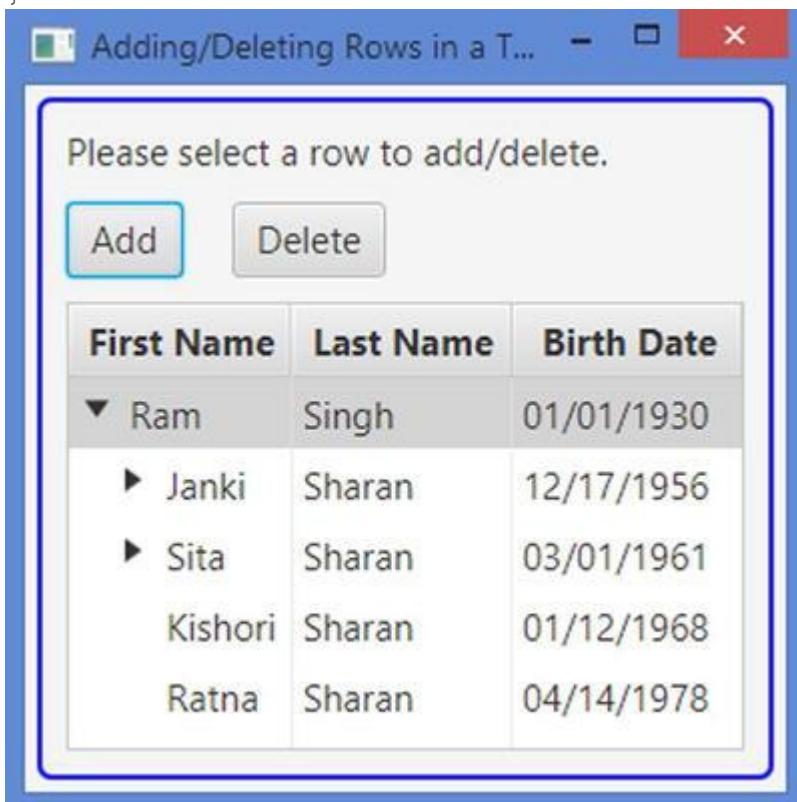
private void deleteRow() {
    // Get the selection model
    TreeTableViewSelectionModel<Person> sm
= treeTable.getSelectionModel();
    if (sm.isEmpty()) {
        System.out.println("Select a row to delete.");
        return;
    }

    int rowIndex = sm.getSelectedIndex();
    TreeItem<Person> selectedItem
= sm.getModelItem(rowIndex);

    TreeItem<Person> parent = selectedItem.getParent();
    if (parent != null) {

```

```
        parent.getChildren().remove(selectedItem);
    } else {
        // Must be deleting the root item
        treeTable.setRoot(null);
    }
}
```



**Figure 15-9.** A TreeTableView allowing users to add or delete rows

## Scrolling in a *TreeTableView*

`TreeTableView` automatically provides vertical and horizontal scrollbars when rows or columns fall beyond the available space. Users can use the scrollbars to scroll to a specific row or column. Sometimes you may need to add programmatic support for scrolling. For example, when you append a row to a `TreeTableView`, you may want to scroll the new row to the view. The `TreeTableView` class contains three methods that can be used to scroll to a specific row or column:

- `scrollTo(int rowIndex)`
  - `scrollToColumn(TreeTableColumn<S, ?> column)`
  - `scrollToColumnIndex(int columnIndex)`

The `scrollTo()` method scrolls the row with the specified `rowIndex` to the view.

The `scrollToColumn()` and `scrollToColumnIndex()` methods scroll to the specified column and `columnIndex`, respectively.

`TreeTableView` fires a `ScrollToEvent` when there is a request to scroll to a row or column using one of the above-mentioned scrolling methods. The `ScrollToEvent` class contains a `getScrollTarget()` method that returns the row index or the column reference, depending on the scroll type, as shown in the following code:

```
TreeTableView<Person> treeTable = ...

// Add a ScrollToEvent for row scrolling
treeTable.setOnScrollTo(e -> {
    int rowIndex = e.getScrollTarget();
    System.out.println("Scrolled to row " + rowIndex);
});

// Add a ScrollToEvent for column scrolling
treeTable.setOnScrollToColumn(e -> {
    TreeTableColumn<Person, ?> column = e.getScrollTarget();
    System.out.println("Scrolled to column "
+ column.getText());
});
```

**Tip** The `ScrollToEvent` is not fired when the user scrolls through the rows and columns. It is fired when you call one of the scrolling-related methods of the `TreeTableView` class.

## Styling `TreeTableView` with CSS

You can style a `TreeTableView` and all its parts, for example, column headers, cells, and placeholders. Applying the CSS to `TreeTableView` is very complex and broad in scope. This section covers a brief overview of CSS styling for `TreeTableView`. The default CSS style-class name for a `TreeTableView` is `tree-table-view`. The default CSS style-class names for a cell, a row, and a column header are `tree-table-cell`, `tree-table-row-cell`, and `column-header`, respectively. The following code shows how to set the font for cells and set the font size and text color for column headers in a `TreeTableView`:

```
/* Set the font for the cells */
.tree-table-row-cell {
    -fx-font-size: 10pt;
    -fx-font-family: Arial;
}

/* Set the font size and text color for column headers */
.tree-table-view .column-header .label {
    -fx-font-size: 10pt;
    -fx-text-fill: blue;
}
```

`TreeTableView` supports the following CSS pseudo-classes:

- `cell-selection`
- `row-selection`
- `constrained-resize`

The `cell-selection` pseudo-class is applied when the cell-level selection is enabled, whereas the `row-selection` pseudo-class is applied for row-level selection. The `constrained-resize` pseudo-class is applied when the column resize policy is `CONSTRAINED_RESIZE_POLICY`.

You can also set the shape of the disclosure node in CSS using the SVG path. The following styles set plus and minus signs as the disclosure nodes for expanded and collapsed nodes, respectively:

```
tree-table-row-cell .tree-disclosure-node .arrow {
    -fx-shape: "M0 -0.5 h2 v2 h1 v-2 h2 v-1 h-2 v-2 h-1 v2 h-2
v1z";
}

.tree-table-row-cell:expanded .tree-disclosure-node .arrow {
    -fx-shape: "M0 -0.5 h5 v-1 h-5 v1z";
    -fx-padding: 4 0.25 4 0.25;
}
```

A `TreeTableView` contains all substructures of a `TableView`. Please refer to the discussion on styling a `TableView` with CSS and `Modena.css` for more details.

## Summary

The `TreeTableView` control combines the features of the `TableView` and `TreeView` controls. It displays a `TreeView` inside a `TableView`. A `TreeView` is used to view hierarchical data. A `TableView` is used to view tabular data. A `TreeTableView` is used to view hierarchical data in a tabular form. `TreeTableView` can be thought of as a nested table or a drill-down table.

An instance of the `TreeTableColumn` class represents a column in a `TreeTableView`. The `getColumns()` method of the `TreeTableView` class returns an `ObservableList` of `TreeTableColumns`, which are the columns added to the `TreeTableView`. You need to add columns to this columns list.

`TreeItems` act as models in a `TreeView`. Each node in the `TreeView` derives its data from the corresponding `TreeItem`. Recall that you can visualize each node (or `TreeItem`) in a `TreeView` as a row with only one column. An `ObservableList` provides the model

in a TableView. Each item in the observable list provides data for a row in the TableView. A TableView can have multiple columns. TreeTableView also uses models for its data. Because it is a combination of TreeView and TableView, it has to decide which type of model it uses. It uses the model based on TreeView. That is, each row in a TreeTableView is defined by a TreeItem in a TreeView. TreeTableView supports multiple columns. Data for columns in a row are derived from the TreeItem for that row.

An instance of the TreeTableView represents a TreeTableView control. The class takes a generic type argument, which is the type of the item contained in the TreeItems. Recall that TreeItems provide the model for a TreeTableView. The generic type of the controls and its TreeItems are the same.

The TreeTableView class provides two constructors. The default constructor creates a TreeTableView with no data. The control displays a placeholder, similar to the one shown by TableView. Like a TableView, TreeTableView contains a placeholder property, which is Node, and if you need to, you can supply your own placeholder. You can add columns and data to a TreeTableView.

TreeTableView supports sorting the same way TableView supports sorting.

Showing and hiding columns in a TreeTableView work the same way they do for TableView. By default, all columns in a TreeTableView are visible. The TreeTableColumn class has a visible property to set the visibility of a column. If you turn off the visibility of a parent column, a column with nested columns, all its nested columns will be invisible.

TreeTableView lets you customize rendering of its cells, using different selection models for its cells and rows. It also allows editing data in its cells and adding and deleting rows. You can also style TreeTableView using CSS.

The next chapter will discuss how to use the 2d Shapes

## CHAPTER 17



### Understanding 2D Shapes

In this chapter, you will learn:

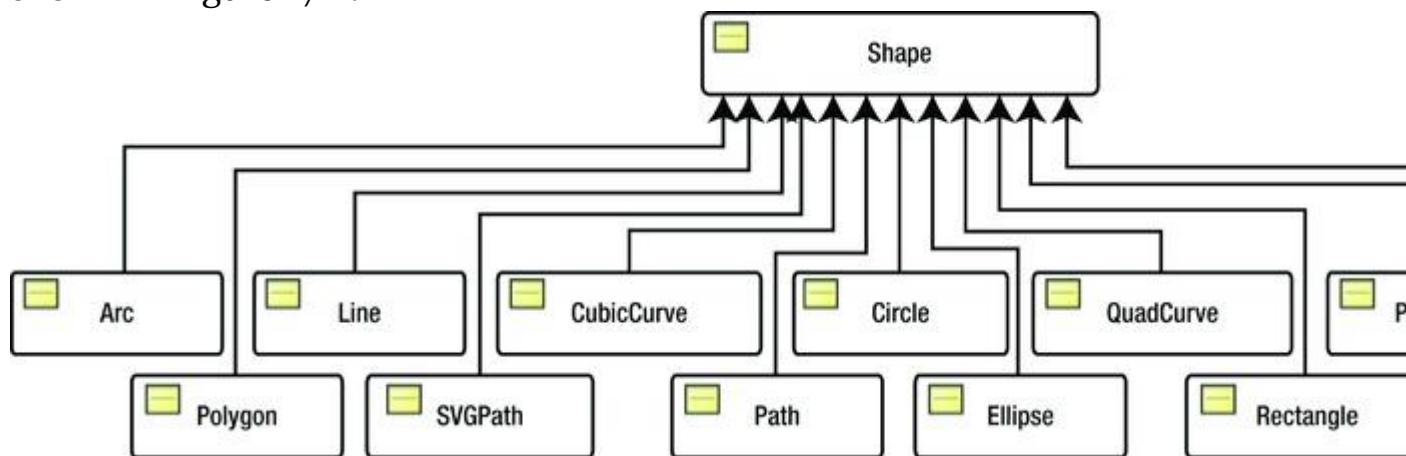
- What 2D shapes are and how they are represented in JavaFX
- How to draw 2D shapes
- How to draw complex shapes using the `Path` class
- How to draw shapes using the Scalable Vector Graphics (SVG)
- How to combine shapes to build another shape
- How to use strokes for a shape
- How to style shapes using Cascading Style Sheets (CSS)

### What Are 2D Shapes?

Any shape that can be drawn in a two-dimensional plane is called a 2D shape. JavaFX offers variety nodes to draw different types of shapes (lines, circles, rectangles, etc.). You can add shapes to a scene graph.

Shapes can be two-dimensional or three-dimensional. In this chapter, I will discuss 2D shapes. Chapter 19 discusses 3D shapes.

All shape classes are in the `javafx.scene.shape` package. Classes representing 2D shapes are inherited from the abstract `Shape` class as shown in Figure 17-1.



**Figure 17-1.** A class diagram for classes representing 2D shapes

A shape has a size and a position, which are defined by their properties. For example, the `width` and `height` properties define the

size of a rectangle; the `radius` property defines the size of a circle, the `x` and `y` properties define the position of the upper-left corner of a rectangle, the `centerX` and `centerY` properties define the center of a circle, etc.

Shapes are not resized by their parents during layout. The size of a shape changes only when its size-related properties are changed. You may find a phrase like “JavaFX shapes are non-resizable.” It means shapes are non-resizable by their parent during layout. They can be resized only by changing their properties.

Shapes have an interior and a stroke. The properties for defining the interior and stroke of a shape are declared in the `Shape` class.

The `fill` property specifies the color to fill the interior of the shape. The default fill is `Color.BLACK`. The `stroke` property specifies the color for the outline stroke, which is `null` by default, except for `Line`, `Polyline`, and `Path`, which have `Color.BLACK` as the default `stroke`. The `strokeWidth` property specifies the width of the outline, which is `1.0px` by default. The `Shape` class contains other stroke-related properties that I will discuss in the section “Understanding the Stroke of a Shape”.

The `Shape` class contains a `smooth` property, which is true by default. Its true value indicates that an antialiasing hint should be used to render the shape. If it is set to false, the antialiasing hint will not be used, which may result in the edges of shapes being not crisp.

The program in Listing 17-1 creates two circles. The first circle has a light gray fill and no stroke, which is the default. The second circle has a yellow fill and a `2.0px` wide black stroke. Figure 17-2 shows the two circles.

### ***Listing 17-1.*** Using fill and stroke Properties of the Shape Class

```
// ShapeTest.java
package com.jdojo.shape;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class ShapeTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
```

```

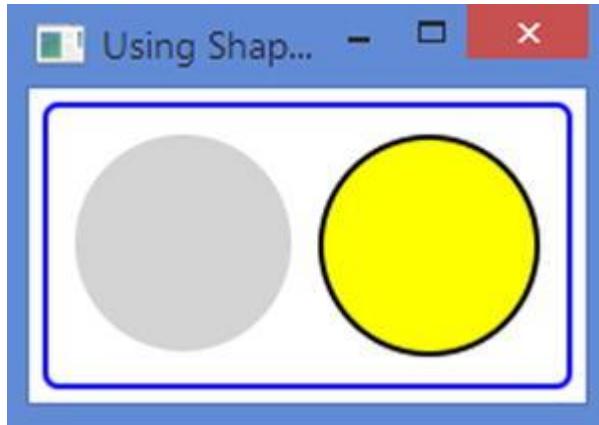
        public void start(Stage stage) {
            // Create a circle with a light gray fill and no
            stroke
            Circle c1 = new Circle(40, 40, 40);
            c1.setFill(Color.LIGHTGRAY);

            // Create a circle with an yellow fill and a black
            stroke of 2.0px
            Circle c2 = new Circle(40, 40, 40);
            c2.setFill(Color.YELLOW);
            c2.setStroke(Color.BLACK);
            c2.setStrokeWidth(2.0);

            HBox root = new HBox(c1, c2);
            root.setSpacing(10);
            root.setStyle("-fx-padding: 10;" +
                "-fx-border-style: solid inside;" +
                "-fx-border-width: 2;" +
                "-fx-border-insets: 5;" +
                "-fx-border-radius: 5;" +
                "-fx-border-color: blue;");

            Scene scene = new Scene(root);
            stage.setScene(scene);
            stage.setTitle("Using Shapes");
            stage.show();
        }
    }
}

```



**Figure 17-2.** Two circles with different fills and strokes

## Drawing 2D Shapes

The following sections describe in detail how to use the JavaFX classes representing 2D shapes to draw those shapes.

### Drawing Lines

An instance of the `Line` class represents a line node. A `Line` has no interior. By default, its `fill` property is set to `null`. Setting `fill` has no

effects. The default `stroke` is `Color.BLACK` and the default `strokeWidth` is `1.0`. The `Line` class contains four double properties.

- `startX`
- `startY`
- `endX`
- `endY`

The `Line` represents a line segment between `(startX, startY)` and `(endX, endY)` points. The `Line` class has a no-args constructor, which defaults all its four properties to zero resulting in a line from `(0, 0)` to `(0, 0)`, which represents a point. Another constructor takes values for `startX`, `startY`, `endX`, and `endY`. After you create a `Line`, you can change its location and length by changing any of the four properties.

The program in Listing 17-2 creates some `Lines` and sets their `stroke` and `strokeWidth` properties. The first `Line` will appear as a point. Figure 17-3 shows the line.

### ***Listing 17-2.*** Using the Line Class to Create Line Nodes

```
// LineTest.java
package com.jdojo.shape;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Line;
import javafx.stage.Stage;

public class LineTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // It will be just a point at (0, 0)
        Line line1 = new Line();

        Line line2 = new Line(0, 0, 50, 0);
        line2.setStrokeWidth(1.0);

        Line line3 = new Line(0, 50, 50, 0);
        line3.setStrokeWidth(2.0);
        line3.setStroke(Color.RED);
    }
}
```

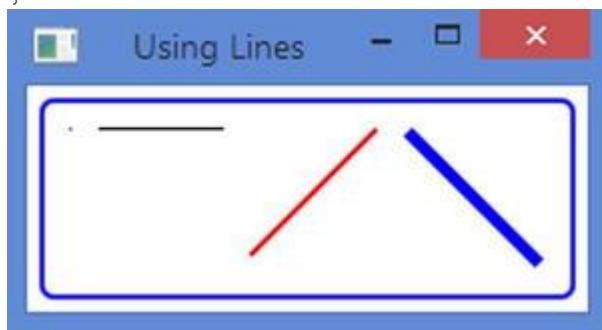
```

        Line line4 = new Line(0, 0, 50, 50);
        line4.setStrokeWidth(5.0);
        line4.setStroke(Color.BLUE);

        HBox root = new HBox(line1, line2, line3, line4);
        root.setSpacing(10);
        root.setStyle("-fx-padding: 10;" +
                      "-fx-border-style: solid inside;" +
                      "-fx-border-width: 2;" +
                      "-fx-border-insets: 5;" +
                      "-fx-border-radius: 5;" +
                      "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using Lines");
        stage.show();
    }
}

```



**Figure 17-3.** Using line nodes

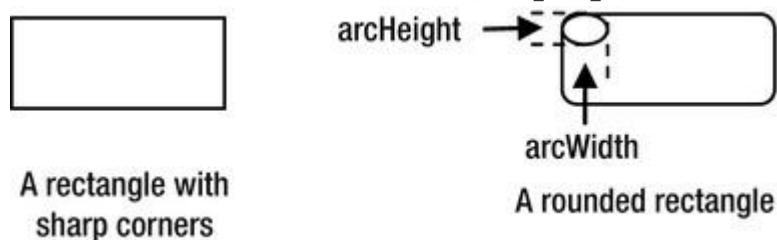
### Drawing Rectangles

An instance of the `Rectangle` class represents a rectangle node. The class uses six properties to define the rectangle.

- `x`
- `y`
- `width`
- `height`
- `arcWidth`
- `arcHeight`

The `x` and `y` properties are the `x` and `y` coordinates of the upper-left corner of the rectangle in the local coordinate system of the node. The `width` and `height` properties are the width and height of the rectangle, respectively. Specify the same width and height to draw a square.

By default, the corners of a rectangle are sharp. A rectangle can have rounded corners by specifying the `arcWidth` and `arcHeight` properties. You can think of one of the quadrants of an ellipse positioned at the four corners to make them round. The `arcWidth` and `arcHeight` properties are the horizontal and vertical diameters of the ellipse. By default, their values are zero, which makes a rectangle have sharp corners. Figure 17-4 shows two rectangles—one with sharp corners and one with rounded corners. The ellipse is shown to illustrate the relationship between the `arcWidth` and `arcHeight` properties for a rounded rectangle.



**Figure 17-4.** Rectangles with sharp and rounded corners

The `Rectangle` class contains several constructors. They take various properties as arguments. The default values for `x`, `y`, `width`, `height`, `arcWidth`, and `arcHeight` properties are zero. The constructors are

- `Rectangle()`
- `Rectangle(double width, double height)`
- `Rectangle(double x, double y, double width, double height)`
- `Rectangle(double width, double height, Paint fill)`

You will not see effects of specifying the values for the `x` and `y` properties for a `Rectangle` when you add it to most of the layout panes as they place their children at (0, 0). A `Pane` uses these properties. The program in Listing 17-3 adds two rectangles to a `Pane`. The first rectangle uses the default values of zero for the `x` and `y` properties. The second rectangle specifies 120 for the `x` property and 20 for the `y` property. Figure 17-5 shows the positions of the two rectangles inside the `Pane`. Notice that the upper-left corner of the second rectangle (on the right) is at (120, 20).

### **Listing 17-3.** Using the Rectangle Class to Create Rectangle Nodes

```
// RectangleTest.java
package com.jdojo.shape;

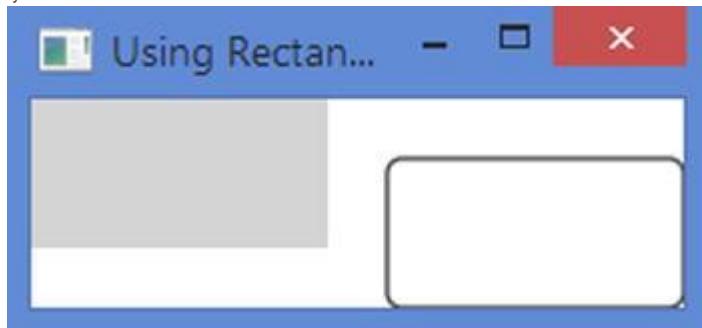
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

public class RectangleTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // x=0, y=0, width=100, height=50, fill=LIGHTGRAY,
        stroke=null
        Rectangle rect1 = new Rectangle(100, 50,
Color.LIGHTGRAY);

        // x=120, y=20, width=100, height=50, fill=WHITE,
        stroke=BLACK
        Rectangle rect2 = new Rectangle(120, 20, 100, 50);
        rect2.setFill(Color.WHITE);
        rect2.setStroke(Color.BLACK);
        rect2.setArcWidth(10);
        rect2.setArcHeight(10);

        Pane root = new Pane();
        root.getChildren().addAll(rect1, rect2);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using Rectangles");
        stage.show();
    }
}
```



**Figure 17-5.** Rectangles inside a Pane, which uses the x and y properties to position them

## Drawing Circles

An instance of the `Circle` class represents a circle node. The class uses three properties to define the circle.

- `centerX`
- `centerY`
- `radius`

The `centerX` and `centerY` properties are the x and y coordinates of the center of the circle in the local coordinate system of the node.

The `radius` property is the radius of the circle. The default values for these properties are zero.

The `Circle` class contains several constructors.

- `Circle()`
- `Circle(double radius)`
- `Circle(double centerX, double centerY, double radius)`
- `Circle(double centerX, double centerY, double radius, Paint fill)`
- `Circle(double radius, Paint fill)`

The program in Listing 17-4 adds two circles to an `HBox`. Notice that the `HBox` does not use `centerX` and `centerY` properties of the circles. Add them to a `Pane` to see the effects. Figure 17-6 shows the two circles.

#### ***Listing 17-4.*** Using the Circle Class to Create Circle Nodes

```
// CircleTest.java
package com.jdojo.shape;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class CircleTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // centerX=0, centerY=0, radius=40, fill=LIGHTGRAY,
        stroke=null
        Circle c1 = new Circle(0, 0, 40);
        c1.setFill(Color.LIGHTGRAY);
    }
}
```

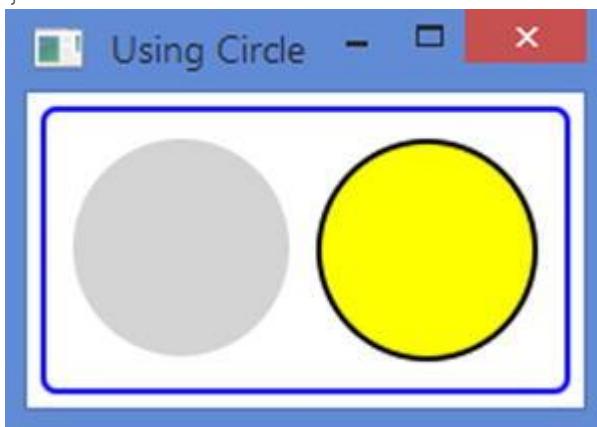
```

        // centerX=10, centerY=10, radius=40. fill=YELLOW,
stroke=BLACK
        Circle c2 = new Circle(10, 10, 40, Color.YELLOW);
        c2.setStroke(Color.BLACK);
        c2.setStrokeWidth(2.0);

        HBox root = new HBox(c1, c2);
        root.setSpacing(10);
        root.setStyle("-fx-padding: 10;" +
                      "-fx-border-style: solid inside;" +
                      "-fx-border-width: 2;" +
                      "-fx-border-insets: 5;" +
                      "-fx-border-radius: 5;" +
                      "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using Circle");
        stage.show();
    }
}

```



**Figure 17-6.** Using circle nodes

### Drawing Ellipses

An instance of the `Ellipse` class represents an ellipse node. The class uses four properties to define the ellipse.

- `centerX`
- `centerY`
- `radiusX`
- `radiusY`

The `centerX` and `centerY` properties are the x and y coordinates of the center of the circle in the local coordinate system of the node. The `radiusX` and `radiusY` are the radii of the ellipse in the horizontal and vertical directions. The default values for these properties are zero. A

circle is a special case of an ellipse when `radiusX` and `radiusY` are the same.

The `Ellipse` class contains several constructors.

- `Ellipse()`
- `Ellipse(double radiusX, double radiusY)`
- `Ellipse(double centerX, double centerY, double radiusX, double radiusY)`

The program in Listing 17-5 creates three instances of the `Ellipse` class. The third instance draws a circle as the program sets the same value for the `radiusX` and `radiusY` properties. Figure 17-7 shows the three ellipses.

### ***Listing 17-5.*** Using the Ellipse Class to Create Ellipse Nodes

```
// EllipseTest.java
package com.jdojo.shape;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Ellipse;
import javafx.stage.Stage;

public class EllipseTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Ellipse e1 = new Ellipse(50, 30);
        e1.setFill(Color.LIGHTGRAY);

        Ellipse e2 = new Ellipse(60, 30);
        e2.setFill(Color.YELLOW);
        e2.setStroke(Color.BLACK);
        e2.setStrokeWidth(2.0);

        // Draw a circle using the Ellipse class
        (radiusX=radiusY=30)
        Ellipse e3 = new Ellipse(30, 30);
        e3.setFill(Color.YELLOW);
        e3.setStroke(Color.BLACK);
        e3.setStrokeWidth(2.0);

        HBox root = new HBox(e1, e2, e3);
        root.setSpacing(10);
        root.setStyle("-fx-padding: 10;" +

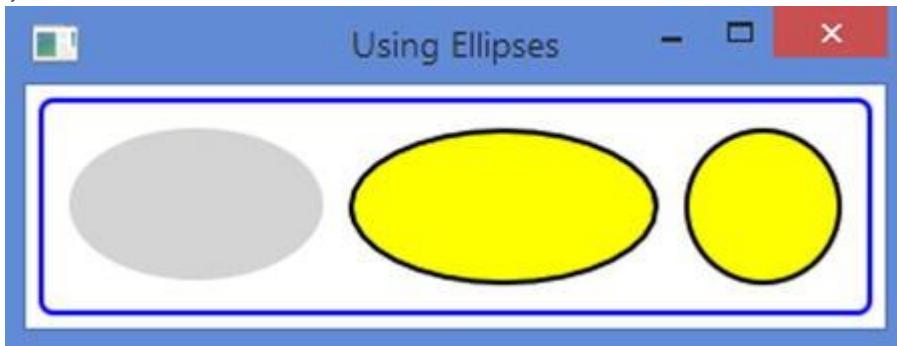
```

```

"-fx-border-style: solid inside;" +
"-fx-border-width: 2;" +
"-fx-border-insets: 5;" +
"-fx-border-radius: 5;" +
"-fx-border-color: blue;");

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("Using Ellipses");
stage.show();
}
}

```



**Figure 17-7.** Using ellipse nodes

## Drawing Polygons

An instance of the `Polygon` class represents a polygon node. The class does not define any public properties. It lets you draw a polygon using an array of (x, y) coordinates defining the vertices of the polygon. Using the `Polygon` class, you can draw any type of geometric shape that is created using connected lines (triangles, pentagons, hexagons, parallelograms, etc.).

The `Polygon` class contains two constructors.

- `Polygon()`
- `Polygon(double... points)`

The no-args constructor creates an empty polygon. You need add the (x, y) coordinates of the vertices of the shape. The polygon will draw a line from the first vertex to the second vertex, from the second to the third, and so on. Finally, the shape is closed by drawing a line from the last vertex to the first vertex.

The `Polygon` class stores the coordinates of the vertices in an `ObservableList<Double>`. You can get the reference of the observable list using the `getPoints()` method. Notice that it stores the coordinates in a list of `Double`, which is simply a number. It is your job to pass the numbers in pairs, so they can be used as (x, y) coordinates of

vertices. If you pass an odd number of numbers, no shape is created. The following snippet of code creates two triangles—one passes the coordinates of the vertices in the constructor and another adds them to the observable list later. Both triangles are geometrically the same.

```
// Create an empty triangle and add vertices later
Polygon triangle1 = new Polygon();
triangle1.getPoints().addAll(50.0, 0.0,
                            0.0, 100.0,
                            100.0, 100.0);

// Create a triangle with vertices
Polygon triangle2 = new Polygon(50.0, 0.0,
                                0.0, 100.0,
                                100.0, 100.0);
```

The program in Listing 17-6 creates a triangle, a parallelogram, and a hexagon using the `Polygon` class as shown in Figure 17-8.

### ***Listing 17-6.*** Using the `Polygon` Class to Create a Triangle, a Parallelogram, and a Hexagon

```
// PolygonTest.java
package com.jdojo.shape;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Polygon;
import javafx.stage.Stage;

public class PolygonTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Polygon triangle1 = new Polygon();
        triangle1.getPoints().addAll(50.0, 0.0,
                                    0.0, 50.0,
                                    100.0, 50.0);
        triangle1.setFill(Color.WHITE);
        triangle1.setStroke(Color.RED);

        Polygon parallelogram = new Polygon();
        parallelogram.getPoints().addAll(30.0, 0.0,
                                         130.0, 0.0,
                                         100.0, 50.0,
                                         0.0, 50.0);
        parallelogram.setFill(Color.YELLOW);
        parallelogram.setStroke(Color.BLACK);
    }
}
```

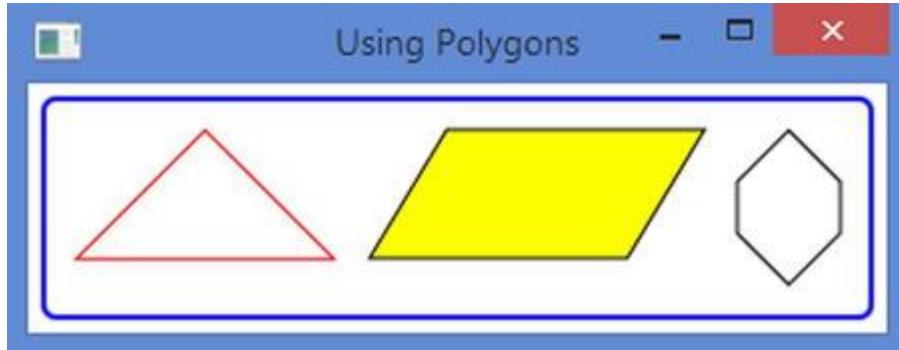
```

        Polygon hexagon = new Polygon(100.0, 0.0,
                                      120.0, 20.0,
                                      120.0, 40.0,
                                      100.0, 60.0,
                                      80.0, 40.0,
                                      80.0, 20.0);
        hexagon.setFill(Color.WHITE);
        hexagon.setStroke(Color.BLACK);

        HBox root = new HBox(triangle1, parallelogram,
hexagon);
        root.setSpacing(10);
        root.setStyle("-fx-padding: 10;" +
                      "-fx-border-style: solid inside;" +
                      "-fx-border-width: 2;" +
                      "-fx-border-insets: 5;" +
                      "-fx-border-radius: 5;" +
                      "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using Polygons");
        stage.show();
    }
}

```



**Figure 17-8.** Using polygon nodes

### Drawing Polylines

A polyline is similar to a polygon, except that it does not draw a line between the last and first points. That is, a polyline is an open polygon. However, the `fill` color is used to fill the entire shape as if the shape was closed.

An instance of the `Polyline` class represents a polyline node. The class does not define any public properties. It lets you draw a polyline using an array of (x, y) coordinates defining the vertices of the polyline. Using the `Polyline` class, you can draw any type of geometric shape that is created using connected lines (triangles, pentagons, hexagons, parallelograms, etc.).

The `Polyline` class contains two constructors.

- `Polyline()`
- `Polyline(double... points)`

The no-args constructor creates an empty polyline. You need add (x, y) coordinates of the vertices of the shape. The polygon will draw a line from the first vertex to the second vertex, from the second to the third, and so on. Unlike a `Polygon`, the shape is not closed automatically. If you want to close the shape, you need to add the coordinates of the first vertex as the last pair of numbers.

If you want to add coordinates of vertices later, add them to the `ObservableList<Double>` returned by the `getPoints()` method of the `Polyline` class. The following snippet of code creates two triangles with the same geometrical properties using different methods. Notice that the first and the last pairs of numbers are the same in order to close the triangle.

```
// Create an empty triangle and add vertices later
Polygon triangle1 = new Polygon();
triangle1.getPoints().addAll(50.0, 0.0,
                           0.0, 100.0,
                           100.0, 100.0,
                           50.0, 0.0);

// Create a triangle with vertices
Polygon triangle2 = new Polygon(50.0, 0.0,
                               0.0, 100.0,
                               100.0, 100.0,
                               50.0, 0.0);
```

The program in Listing 17-7 creates a triangle, an open parallelogram, and a hexagon using the `Polyline` class as shown in Figure 17-9.

***Listing 17-7.*** Using the `Polyline` Class to Create a Triangle, an Open Parallelogram, and a Hexagon

```
// PolylineTest.java
package com.jdojo.shape;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Polyline;
import javafx.stage.Stage;

public class PolylineTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
```

```
}

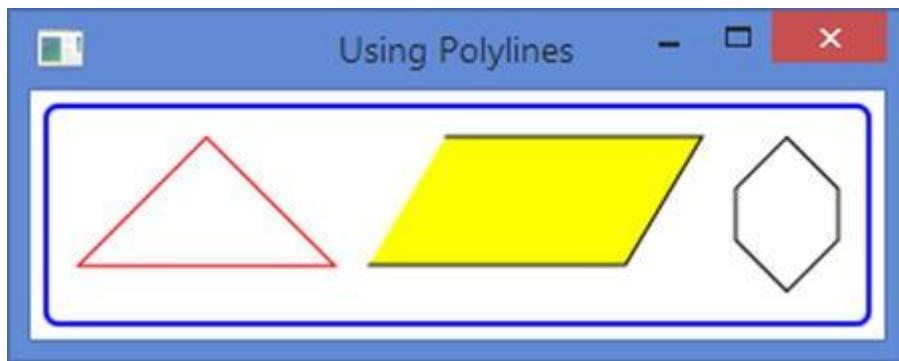
@Override
public void start(Stage stage) {
    Polyline triangle1 = new Polyline();
    triangle1.getPoints().addAll(50.0, 0.0,
                                0.0, 50.0,
                                100.0, 50.0,
                                50.0, 0.0);
    triangle1.setFill(Color.WHITE);
    triangle1.setStroke(Color.RED);

    // Create an open parallelogram
    Polyline parallelogram = new Polyline();
    parallelogram.getPoints().addAll(30.0, 0.0,
                                    130.0, 0.0,
                                    100.0, 50.0,
                                    0.0, 50.0);
    parallelogram.setFill(Color.YELLOW);
    parallelogram.setStroke(Color.BLACK);

    Polyline hexagon = new Polyline(100.0, 0.0,
                                    120.0, 20.0,
                                    120.0, 40.0,
                                    100.0, 60.0,
                                    80.0, 40.0,
                                    80.0, 20.0,
                                    100.0, 0.0);
    hexagon.setFill(Color.WHITE);
    hexagon.setStroke(Color.BLACK);

    HBox root = new HBox(triangle1, parallelogram,
hexagon);
    root.setSpacing(10);
    root.setStyle("-fx-padding: 10;" +
                  "-fx-border-style: solid inside;" +
                  "-fx-border-width: 2;" +
                  "-fx-border-insets: 5;" +
                  "-fx-border-radius: 5;" +
                  "-fx-border-color: blue;");

    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("Using Polylines");
    stage.show();
}
}
```



**Figure 17-9.** Using polyline nodes

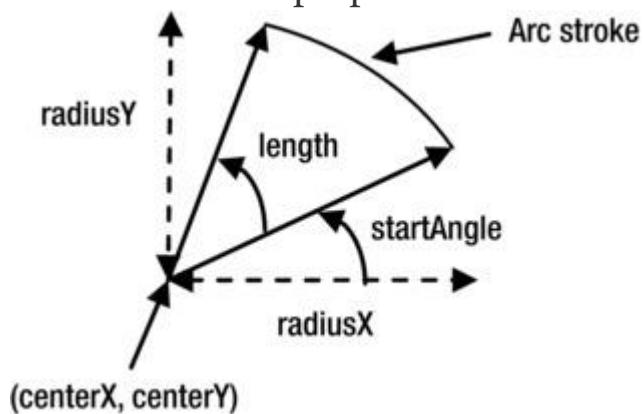
### Drawing Arcs

An instance of the `Arc` class represents a sector of an ellipse. The class uses seven properties to define the ellipse.

- `centerX`
- `centerY`
- `radiusX`
- `radiusY`
- `startAngle`
- `length`
- `type`

The first four properties define an ellipse. Please refer to the section “Drawing Ellipses” for how to define an ellipse. The last three properties define a sector of the ellipse that is the `Arc` node.

The `startAngle` property specifies the start angle of the section in degrees measured counterclockwise from the positive x-axis. It defines the beginning of the arc. The `length` is an angle in degrees measured counterclockwise from the start angle to define the end of the sector. If the `length` property is set to 360, the `Arc` is a full ellipse. Figure 17-10 illustrates the properties.

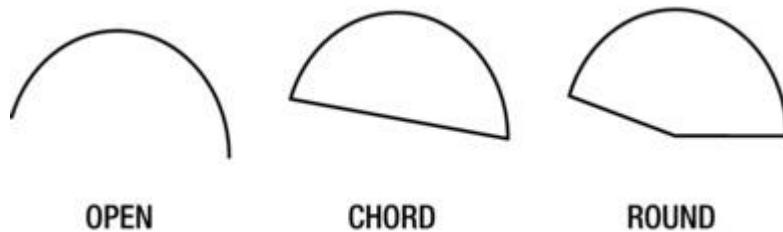


**Figure 17-10.** Properties defining an Arc

The type property specifies the way the `Arc` is closed. It is one of the constants, `OPEN`, `CHORD`, and `ROUND`, defined in the `ArcType` enum.

- The `ArcType.OPEN` does not close the arc.
- The `ArcType.CHORD` closes the arc by joining the starting and ending points by a straight line.
- The `ArcType.ROUND` closes the arc by joining the starting and ending point to the center of the ellipse.

Figure 17-11 shows the three closure types for an arc. The default type for an `Arc` is `ArcType.OPEN`. If you do not apply a stroke to an `Arc`, both `ArcType.OPEN` and `ArcType.CHORD` look the same.



**Figure 17-11.** Closure types of an arc

The `Arc` class contains two constructors:

- `Arc()`
- `Arc(double centerX, double centerY, double radiusX, double radiusY, double startAngle, double length)`

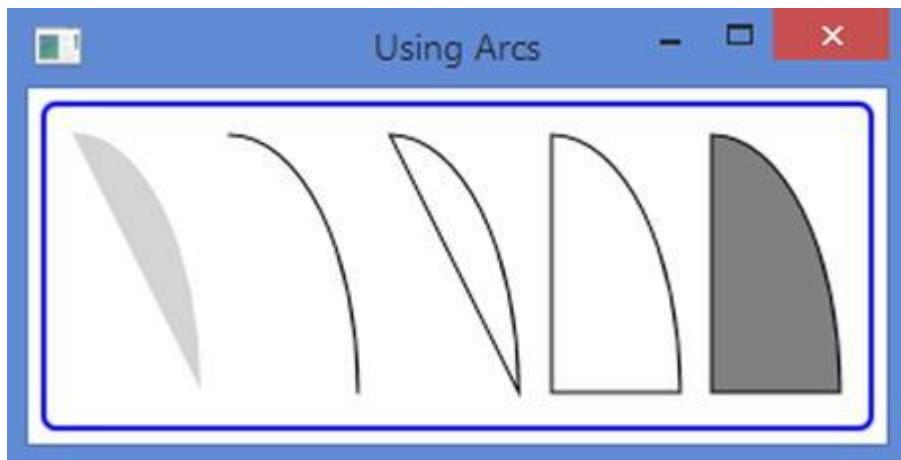
The program in Listing 17-8 shows how to create `Arc` nodes. The resulting window is shown in Figure 17-12.

### **Listing 17-8.** Using the Arc Class to Create Arcs, Which Are Sectors of Ellipses

```
// ArcTest.java
package com.jdojo.shape;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Arc;
import javafx.scene.shape.ArcType;
import javafx.stage.Stage;
```

```
public class ArcTest extends Application {  
    public static void main(String[] args) {  
        Application.launch(args);  
    }  
  
    @Override  
    public void start(Stage stage) {  
        // An OPEN arc with a fill  
        Arc arc1 = new Arc(0, 0, 50, 100, 0, 90);  
        arc1.setFill(Color.LIGHTGRAY);  
  
        // An OPEN arc with no fill and a stroke  
        Arc arc2 = new Arc(0, 0, 50, 100, 0, 90);  
        arc2.setFill(Color.TRANSPARENT);  
        arc2.setStroke(Color.BLACK);  
  
        // A CHORD arc with no fill and a stroke  
        Arc arc3 = new Arc(0, 0, 50, 100, 0, 90);  
        arc3.setFill(Color.TRANSPARENT);  
        arc3.setStroke(Color.BLACK);  
        arc3.setType(ArcType.CHORD);  
  
        // A ROUND arc with no fill and a stroke  
        Arc arc4 = new Arc(0, 0, 50, 100, 0, 90);  
        arc4.setFill(Color.TRANSPARENT);  
        arc4.setStroke(Color.BLACK);  
        arc4.setType(ArcType.ROUND);  
  
        // A ROUND arc with a gray fill and a stroke  
        Arc arc5 = new Arc(0, 0, 50, 100, 0, 90);  
        arc5.setFill(Color.GRAY);  
        arc5.setStroke(Color.BLACK);  
        arc5.setType(ArcType.ROUND);  
  
        HBox root = new HBox(arc1, arc2, arc3, arc4, arc5);  
        root.setSpacing(10);  
        root.setStyle("-fx-padding: 10;" +  
                     "-fx-border-style: solid inside;" +  
                     "-fx-border-width: 2;" +  
                     "-fx-border-insets: 5;" +  
                     "-fx-border-radius: 5;" +  
                     "-fx-border-color: blue;");  
  
        Scene scene = new Scene(root);  
        stage.setScene(scene);  
        stage.setTitle("Using Arcs");  
        stage.show();  
    }  
}
```



**Figure 17-12.** Using Arc nodes

### Drawing Quadratic Curves

Bezier curves are used in computer graphics to draw smooth curves. An instance of the `QuadCurve` class represents a quadratic Bezier curve segment intersecting two specified points using a specified Bezier control point. The `QuadCurve` class contains six properties to specify the three points.

- `startX`
- `startY`
- `controlX`
- `controlY`
- `endX`
- `endY`

The `QuadCurve` class contains two constructors.

- `QuadCurve()`
- `QuadCurve(double startX, double startY, double controlX, double controlY, double endX, double endY)`

The program in Listing 17-9 draws the same quadratic Bezier curve twice—once with a stroke and a transparent fill and once with no stroke and a light gray fill. Figure 17-13 shows the two curves.

### **Listing 17-9.** Using the QuadCurve Class to Draw Quadratic BezierCurve

```
// QuadCurveTest.java
package com.jdojo.shape;
```

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.QuadCurve;
import javafx.stage.Stage;

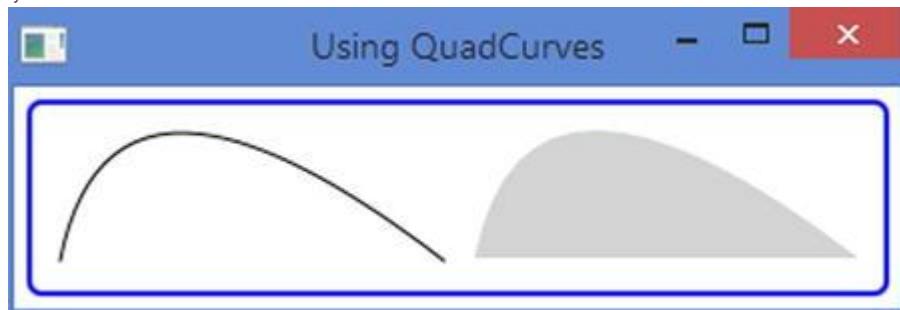
public class QuadCurveTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        QuadCurve qc1 = new QuadCurve(0, 100, 20, 0, 150,
100);
        qc1.setFill(Color.TRANSPARENT);
        qc1.setStroke(Color.BLACK);

        QuadCurve qc2 = new QuadCurve(0, 100, 20, 0, 150,
100);
        qc2.setFill(Color.LIGHTGRAY);

        HBox root = new HBox(qc1, qc2);
        root.setSpacing(10);
        root.setStyle("-fx-padding: 10;" +
"-fx-border-style: solid inside;" +
"-fx-border-width: 2;" +
"-fx-border-insets: 5;" +
"-fx-border-radius: 5;" +
"-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using QuadCurves");
        stage.show();
    }
}
```



**Figure 17-13.** Using quadratic Bezier curves

## Drawing Cubic Curves

An instance of the `CubicCurve` class represents a cubic Bezier curve segment intersecting two specified points using two specified Bezier control points. Please refer to the Wikipedia article at [http://en.wikipedia.org/wiki/Bezier\\_curves](http://en.wikipedia.org/wiki/Bezier_curves) for a detailed explanation and demonstration of Bezier curves. The `CubicCurve` class contains eight properties to specify the four points.

- `startX`
- `startY`
- `controlX1`
- `controlY1`
- `controlX2`
- `controlY2`
- `endX`
- `endY`

The `CubicCurve` class contains two constructors.

- `CubicCurve()`
- `CubicCurve(double startX, double startY, double controlX1, double controlY1, double controlX2, double controlY2, double endX, double endY)`

The program in Listing 17-10 draws the same cubic Bezier curve twice—once with a stroke and a transparent fill and once with no stroke and a light gray fill. Figure 17-14 shows the two curves.

### ***Listing 17-10.*** Using the `CubicCurve` Class to Draw Cubic Bezier Curve

```
// CubicCurveTest.java
package com.jdojo.shape;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.CubicCurve;
import javafx.stage.Stage;

public class CubicCurveTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
```

```

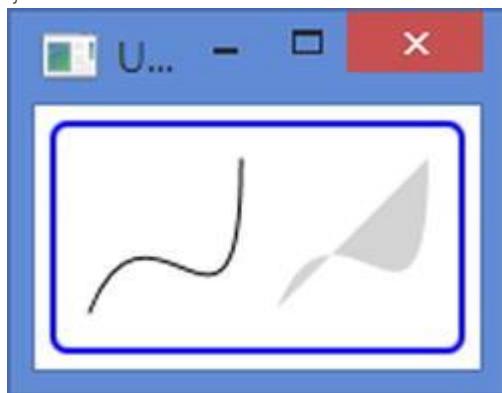
        public void start(Stage stage) {
            CubicCurve cc1 = new CubicCurve(0, 50, 20, 0, 50,
80, 50, 0);
                cc1.setFill(Color.TRANSPARENT);
                cc1.setStroke(Color.BLACK);

            CubicCurve cc2 = new CubicCurve(0, 50, 20, 0, 50,
80, 50, 0);
                cc2.setFill(Color.LIGHTGRAY);

            HBox root = new HBox(cc1, cc2);
            root.setSpacing(10);
            root.setStyle("-fx-padding: 10;" +
                            "-fx-border-style: solid inside;" +
                            "-fx-border-width: 2;" +
                            "-fx-border-insets: 5;" +
                            "-fx-border-radius: 5;" +
                            "-fx-border-color: blue;");

            Scene scene = new Scene(root);
            stage.setScene(scene);
            stage.setTitle("Using CubicCurves");
            stage.show();
        }
    }
}

```



**Figure 17-14.** Using cubic Bezier curves

## Building Complex Shapes Using the Path Class

I discussed several shape classes in the previous sections. They are used to draw simple shapes. It is not convenient to use them for complex shapes. You can draw complex shapes using the `Path` class. An instance of the `Path` class defines the path (outline) of a shape. A path consists of one or more subpaths. A subpath consists of one or more path elements. Each subpath has a starting point and an ending point.

A path element is an instance of the `PathElement` abstract class. The following subclasses of the `PathElement` class exist to represent specific type of path elements:

- MoveTo
- LineTo
- HLineTo
- VLineTo
- ArcTo
- QuadCurveTo
- CubicCurveTo
- ClosePath

Before you see an example, let us outline the process of creating a shape using the `Path` class. The process is similar to drawing a shape on a paper with a pencil. First, you place the pencil on the paper. You can restate it, “You move the pencil to a point on the paper.” Regardless of what shape you want to draw, moving the pencil to a point must be the first step. Now, you start moving your pencil to draw a path element (e.g., a horizontal line). The starting point of the current path element is the same as the ending point of the previous path element. Keep drawing as many path elements as needed (e.g., a vertical line, an arc, and a quadratic Bezier curve). At the end, you can end the last path element at the same point where you started or somewhere else.

The coordinates defining a `PathElement` can be absolute or relative. By default, coordinates are absolute. It is specified by the `absolute` property of the `PathElement` class. If it is true, which is the default, the coordinates are absolute. If it is false, the coordinates are relative. The absolute coordinates are measured relative to the local coordinate system of the node. Relative coordinates are measured treating the ending point of the previous `PathElement` as the origin.

The `Path` class contains three constructors.

- `Path()`
- `Path(Collection<? extends PathElement> elements)`
- `Path(PathElement... elements)`

The no-args constructor creates an empty shape. The other two constructors take a list of path elements as arguments. A `Path` stores path elements in an `ObservableList<PathElement>`. You can get the reference of the list using the `getElements()` method. You can modify the list of path elements to modify the shape. The following snippet of code shows two ways of creating shapes using the `Path` class:

```
// Pass the path elements to the constructor
Path shape1 = new Path(pathElement1, pathElement2, pathElement3);

// Create an empty path and add path elements to the elements
```

```
list
Path shape2 = new Path();
shape2.getElements().addAll(pathElement1, pathElement2,
pathElement3);
```

**Tip** An instance of the `PathElement` may be added as a path element to `Path` objects simultaneously. A `Path` uses the same fill and stroke for all its path elements.

## The MoveTo Path Element

A `MoveTo` path element is used to make the specified x and y coordinates as the current point. It has the effect of lifting and placing the pencil at the specified point on the paper. The first path element of a `Path` object must be a `MoveTo` element and it must not use relative coordinates. The `MoveTo` class defines two `double` properties that are the x and y coordinates of the point.

- X
- Y

The `MoveTo` class contains two constructors. The no-args constructor sets the current point to (0.0, 0.0). The other constructor takes the x and y coordinates of the current point as arguments.

```
// Create a MoveTo path element to move the current point to
(0.0, 0.0)
MoveTo mt1 = new MoveTo();

// Create a MoveTo path element to move the current point to
(10.0, 10.0)
MoveTo mt2 = new MoveTo(10.0, 10.0);
```

**Tip** A path must start with a `MoveTo` path element. You can have multiple `MoveTo` path elements in a path. A subsequent `MoveTo` element denotes the starting point of a new subpath.

## The LineTo Path Element

A `LineTo` path element draws a straight line from the current point to the specified point. It contains two `double` properties that are the x and y coordinates of the end of the line:

- X
- Y

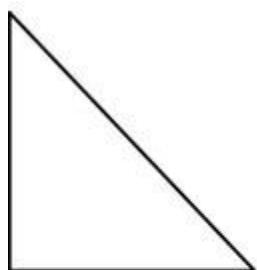
The `LineTo` class contains two constructors. The no-args constructor sets the end of the line to (0.0, 0.0). The other constructor takes the x and y coordinates of the end of the line as arguments.

```
// Create a LineTo path element with its end at (0.0, 0.0)
LineTo lt1 = new LineTo();

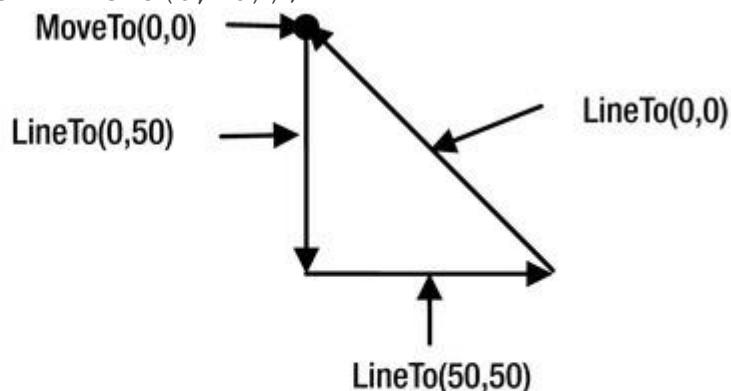
// Create a LineTo path element with its end at (10.0, 10.0)
LineTo lt2 = new LineTo(10.0, 10.0);
```

With the knowledge of the `MoveTo` and `LineTo` path elements, you can construct shapes that are made of lines only. The following snippet of code creates a triangle as shown in Figure 17-15. The figure shows the triangle and its path elements. The arrows show the flow of the drawing. Notice that the drawing starts at (0,0) using the first `MoveTo` path element.

```
Path triangle = new Path(new MoveTo(0, 0),
    new LineTo(0, 50),
    new LineTo(50, 50),
    new LineTo(0, 0));
```



The triangle



The path elements of the triangle

**Figure 17-15.** Creating a triangle using the `MoveTo` and `LineTo` path elements

The `ClosePath` path element closes a path by drawing a straight line from the current point to the starting point of the path. If multiple `MoveTo` path elements exist in a path, a `ClosePath` draws a straight line from the current point to the point identified by the last `MoveTo`. You can rewrite the path for the previous triangle example using a `ClosePath`.

```
Path triangle = new Path(new MoveTo(0, 0),
    new LineTo(0, 50),
    new LineTo(50, 50),
    new ClosePath());
```

The program in Listing 17-11 creates two `Path` nodes: one triangle and one with two inverted triangles to give it a look of a star as shown in Figure 17-16. In the second shape, each triangle is created as a subpath—each subpath starting with a `MoveTo` element. Notice the two uses of the `ClosePath` elements. Each `ClosePath` closes its subpath.

### ***Listing 17-11.*** Using the Path Class to Create a Triangle and a Star

```
// PathTest.java
package com.jdojo.shape;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.shape.ClosePath;
import javafx.scene.shape.LineTo;
import javafx.scene.shape.MoveTo;
import javafx.scene.shape.Path;
import javafx.stage.Stage;

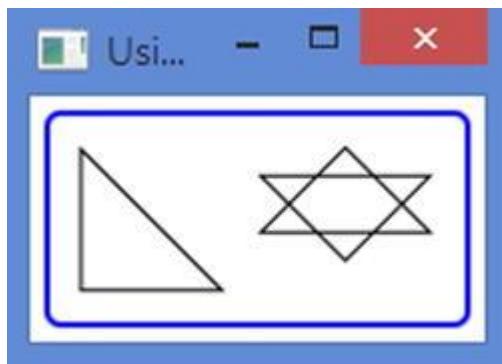
public class PathTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Path triangle = new Path(new MoveTo(0, 0),
            new LineTo(0, 50),
            new LineTo(50, 50),
            new ClosePath());

        Path star = new Path();
        star.getElements().addAll(new MoveTo(30, 0),
            new LineTo(0, 30),
            new LineTo(60, 30),
            new ClosePath(), /* new
LineTo(30, 0), */
            new MoveTo(0, 10),
            new LineTo(60, 10),
            new LineTo(30, 40),
            new ClosePath() /*new LineTo(0,
10) */);

        HBox root = new HBox(triangle, star);
        root.setSpacing(10);
        root.setStyle("-fx-padding: 10;" +
            "-fx-border-style: solid inside;" +
            "-fx-border-width: 2;" +
            "-fx-border-insets: 5;" +
            "-fx-border-radius: 5;" +
            "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using Paths");
        stage.show();
    }
}
```



**Figure 17-16.** Shapes based on path elements

### The HLineTo and VLineTo Path Elements

The `HLineTo` path element draws a horizontal line from the current point to the specified x coordinate. The y coordinate of the ending point of the line is the same as the y coordinate of the current point.

The `x` property of the `HLineTo` class specifies the x coordinate of the ending point.

```
// Create an horizontal line from the current point (x, y) to
// (50, y)
HLineTo hlt = new HLineTo(50);
```

The `VLineTo` path element draws a vertical line from the current point to the specified y coordinate. The x coordinate of the ending point of the line is the same as the x coordinate of the current point.

The `y` property of the `VLineTo` class specifies the y coordinate of the ending point.

```
// Create a vertical line from the current point (x, y) to (x,
// 50)
VLineTo vlt = new VLineTo(50);
```

**Tip** The `LineTo` path element is the generic version of `HLineTo` and `VLineTo`.

The following snippet of code creates the same triangle as discussed in the previous section. This time, you use `HLineTo` and `VLineTo` path elements to draw the base and height sides of the triangle instead of the `LineTo` path elements.

```
Path triangle = new Path(new MoveTo(0, 0),
                        new VLineTo(50),
                        new HLineTo(50),
                        new ClosePath());
```

### The ArcTo Path Element

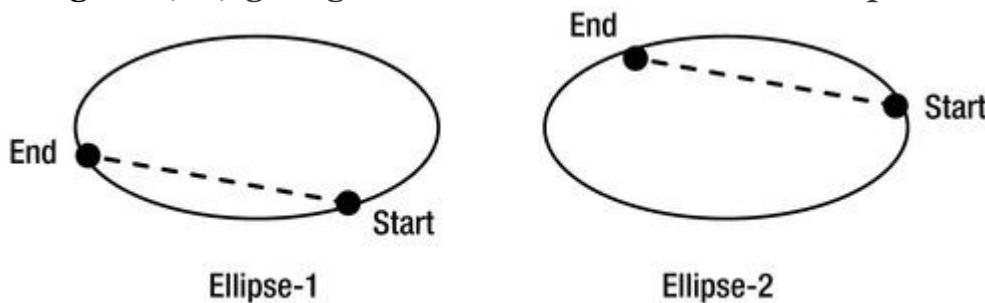
An `ArcTo` path element defines a segment of ellipse connecting the current point and the specified point. It contains the following properties:

- `radiusX`
- `radiusY`
- `x`
- `y`
- `XAxisRotation`
- `largeArcFlag`
- `sweepFlag`

The `radiusX` and `radiusY` properties specify the horizontal and vertical radii of the ellipse. The `x` and `y` properties specify the `x` and `y` coordinates of the ending point of the arc. Note that the starting point of the arc is the current point of the path.

The `XAxisRotation` property specifies the rotation of the `x`-axis of the ellipse in degrees. Note that the rotation is for the `x`-axis of the ellipse from which the arc is obtained, not the `x`-axis of the coordinate system of the node. A positive value rotates the `x`-axis counterclockwise.

The `largeArcFlag` and `sweepFlag` properties are Boolean type, and by default, they are set to false. Their uses need a detailed explanation. Two ellipses can pass through two given points as shown in Figure 17-17 giving us four arcs to connect the two points.



**Figure 17-17.** Effects of the `largeArcFlag` and `sweepFlag` properties on an `ArcTo` path element

Figure 17-17 shows starting and ending points labeled `Start` and `End`, respectively. Two points on an ellipse can be traversed through the larger arc or smaller arc. If the `largeArcFlag` is true, the larger arc is used. Otherwise, the smaller arc is used.

When it is decided that the larger or smaller arc is used, you still have two choices: which ellipse of the two possible ellipses will be used? This is determined by the `sweepFlag` property. Try drawing the arc from the starting point to the point ending point using two selected arcs—the two

larger arcs or the two smaller arcs. For one arc, the traversal will be clockwise and for the other counterclockwise. If the `sweepFlag` is true, the ellipse with the clockwise traversal is used. If the `sweepFlag` is false, the ellipse with the counterclockwise traversal is used. Table 17-1 shows which type of arc from which ellipse will be used based on the two properties.

**Table 17-1.** Choosing the Arc Segment and the Ellipse Based on the `largeArcFlag` and `sweepFlag` Properties

<b>largeArcFlag</b>	<b>sweepFlag</b>	<b>Arc Type</b>	<b>Ellipse</b>
true	true	Larger	Ellipse-2
true	false	Larger	Ellipse-1
false	true	Smaller	Ellipse-1
false	false	Smaller	Ellipse-2

The program in Listing 17-12 uses an `ArcTo` path element to build a `Path` object. The program lets the user change properties of the `ArcTo` path element. Run the program and change `largeArcFlag`, `sweepFlag`, and other properties to see how they affect the `ArcTo` path element.

### **Listing 17-12.** Using ArcTo Path Elements

```
// ArcToTest.java
package com.jdojo.shape;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.CheckBox;
import javafx.scene.control.Label;
import javafx.scene.control.Slider;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.shape.ArcTo;
import javafx.scene.shape.HLineTo;
import javafx.scene.shape.MoveTo;
import javafx.scene.shape.Path;
import javafx.scene.shape.VLineTo;
import javafx.stage.Stage;

public class ArcToTest extends Application {
    private ArcTo arcTo;

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

```

    }

@Override
public void start(Stage stage) {
    // Create the ArcTo path element
    arcTo = new ArcTo();

    // Use the arcTo element to build a Path
    Path path = new Path(new MoveTo(0, 0),
                         new VLineTo(100),
                         new HLineTo(100),
                         new VLineTo(50),
                         arcTo);

    BorderPane root = new BorderPane();
    root.setTop(this.getTopPane());
    root.setCenter(path);
    root.setStyle("-fx-padding: 10;" +
                  "-fx-border-style: solid inside;" +
                  "-fx-border-width: 2;" +
                  "-fx-border-insets: 5;" +
                  "-fx-border-radius: 5;" +
                  "-fx-border-color: blue;");

    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("Using ArcTo Path Elements");
    stage.show();
}

private GridPane getTopPane() {
    CheckBox largeArcFlagCbx = new
    CheckBox("largeArcFlag");
    CheckBox sweepFlagCbx = new CheckBox("sweepFlag");
    Slider xRotationSlider = new Slider(0, 360, 0);
    xRotationSlider.setPrefWidth(300);
    xRotationSlider.setBlockIncrement(30);
    xRotationSlider.setShowTickMarks(true);
    xRotationSlider.setShowTickLabels(true);

    Slider radiusXSlider = new Slider(100, 300, 100);
    radiusXSlider.setBlockIncrement(10);
    radiusXSlider.setShowTickMarks(true);
    radiusXSlider.setShowTickLabels(true);

    Slider radiusYSlider = new Slider(100, 300, 100);
    radiusYSlider.setBlockIncrement(10);
    radiusYSlider.setShowTickMarks(true);
    radiusYSlider.setShowTickLabels(true);

    // Bind ArcTo properties to the control data
    arcTo.largeArcFlagProperty().bind(largeArcFlagCbx.selectedProperty());
    arcTo.sweepFlagProperty().bind(sweepFlagCbx.selectedProperty());
}

```

```

        arcTo.XAxisRotationProperty().bind(xRotationSlider.valueProperty());
        arcTo.radiusXProperty().bind(radiusXSlider.valueProperty());
        arcTo.radiusYProperty().bind(radiusYSlider.valueProperty());

        GridPane pane = new GridPane();
        pane.setHgap(5);
        pane.setVgap(10);
        pane.addRow(0, largeArcFlagCbx, sweepFlagCbx);
        pane.addRow(1, new Label("XAxisRotation"),
xRotationSlider);
        pane.addRow(2, new Label("radiusX"), radiusXSlider);
        pane.addRow(3, new Label("radiusY"), radiusYSlider);

        return pane;
    }
}

```

## The QuadCurveTo Path Element

An instance of the `QuadCurveTo` class draws a quadratic Bezier curve from the current point to the specified ending point (x, y) using the specified control point (`controlX`, `controlY`). It contains four properties to specify the ending and control points.

- `x`
- `y`
- `controlX`
- `controlY`

The `x` and `y` properties specify the `x` and `y` coordinates of the ending point. The `controlX` and `controlY` properties specify the `x` and `y` coordinates of the control point.

The `QuadCurveTo` class contains two constructors.

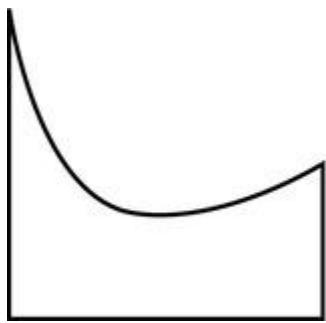
- `QuadCurveTo()`
- `QuadCurveTo(double controlX, double controlY, double x, double y)`

The following snippet of code uses a `QuadCurveTo` with the (10, 100) control point and (0, 0) ending point. Figure 17-18 shows the resulting path.

```

Path path = new Path(new MoveTo(0, 0),
    new VLineTo(100),
    new HLineTo(100),
    new VLineTo(50),
    new QuadCurveTo(10, 100, 0, 0));

```



**Figure 17-18.** Using a `QuadCurveTo` path element

### The CubicCurveTo Path Element

An instance of the `CubicCurveTo` class draws a cubic Bezier curve from the current point to the specified ending point (`x`, `y`) using the specified control points (`controlX1`, `controlY1`) and (`controlX2`, `controlY2`). It contains six properties to specify the ending and control points:

- `x`
- `y`
- `controlX1`
- `controlY1`
- `controlX2`
- `controlY2`

The `x` and `y` properties specify the `x` and `y` coordinates of the ending point. The `controlX1` and `controlY1` properties specify the `x` and `y` coordinates of the first control point.

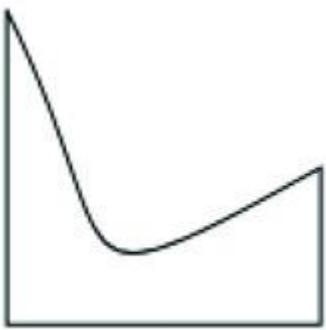
The `controlX2` and `controlY2` properties specify the `x` and `y` coordinates of the second control point.

The `CubicCurveTo` class contains two constructors:

- `CubicCurveTo()`
- `CubicCurveTo(double controlX1, double controlY1, double controlX2, double controlY2, double x, double y)`

The following snippet of code uses a `CubicCurveTo` with the (10, 100) and (40, 80) as control points, and (0, 0) as the ending point. Figure 17-19 shows the resulting path.

```
Path path = new Path(new MoveTo(0, 0),
                     new VLineTo(100),
                     new HLineTo(100),
                     new VLineTo(50),
                     new CubicCurveTo(10, 100, 40, 80, 0, 0));
```



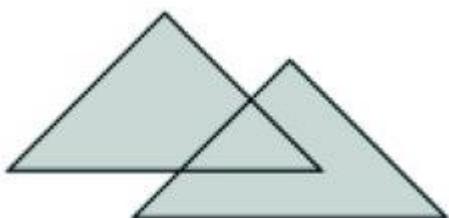
**Figure 17-19.** Using a `QuadCurveTo` path element

### The ClosePath Path Element

The `ClosePath` path element closes the current subpath. Note that a `Path` may consist of multiple subpaths, and, therefore, it is possible to have multiple `ClosePath` elements in a `Path`. A `ClosePath` element draws a straight line from the current point to the initial point of the current subpath and ends the subpath. A `ClosePath` element may be followed by a `MoveTo` element, and in that case, the `MoveTo` element is the starting point of the next subpath. If a `ClosePath` element is followed by a path element other than a `MoveTo` element, the next subpath starts at the starting point of the subpath that was closed by the `ClosePath` element.

The following snippet of code creates a `Path` object, which uses two subpaths. Each subpath draws a rectangle. The subpaths are closed using `ClosePath` elements. Figure 17-20 shows the resulting shape.

```
Path p1 = new Path(new MoveTo(50, 0),
    new LineTo(0, 50),
    new LineTo(100, 50),
    new ClosePath(),
    new MoveTo(90, 15),
    new LineTo(40, 65),
    new LineTo(140, 65),
    new ClosePath());
p1.setFill(Color.LIGHTGRAY);
```

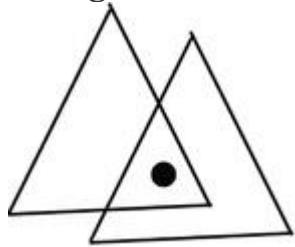


**Figure 17-20.** A shape using two subpaths and `ClosePath` element

### The Fill Rule for a Path

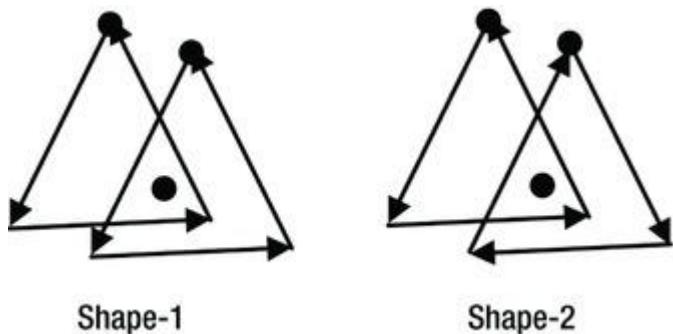
A Path can be used to draw very complex shapes. Sometimes, it is hard to determine whether a point is inside or outside the shape.

The Path class contains a `fillRule` property that is used to determine whether a point is inside a shape. Its value could be one of the constants of the `FillRule` enum: `NON_ZERO` and `EVEN_ODD`. If a point is inside the shape, it will be rendered using the fill color. Figure 17-21 shows two triangles created by a Path and a point in area common to both triangles. I will discuss whether the point is considered inside the shape.



**Figure 17-21.** A shape made of two triangular subpaths

The direction of the stroke is the vital factor in determining whether a point is inside a shape. The shape in Figure 17-21 can be drawn using strokes in different directions. Figure 17-22 shows two of them. In Shape-1, both triangles use counterclockwise strokes. In Shape-2, one triangle uses a counterclockwise stroke and another uses a clockwise stroke.

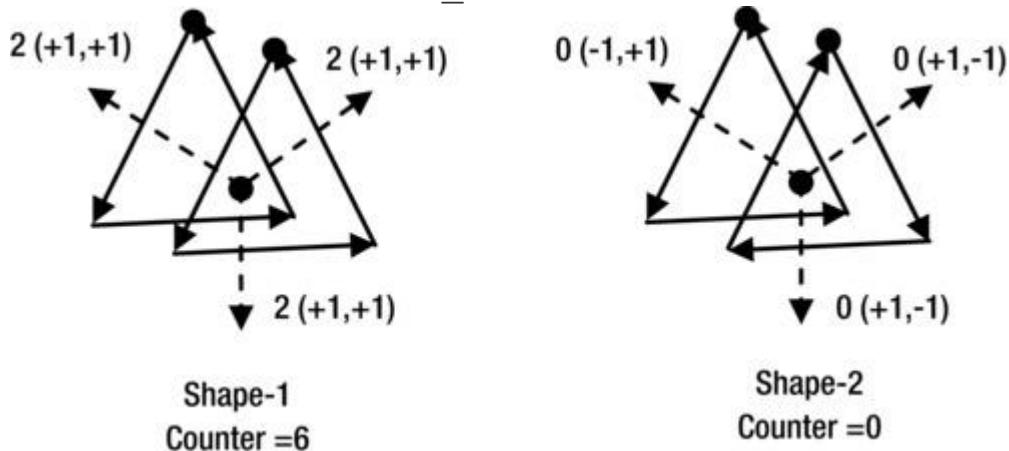


**Figure 17-22.** A shape made of two triangular subpaths using different stroke directions

The fill rule of a Path draws rays from the point to infinity, so they can intersect all path segments. In the `NON_ZERO` fill rule, if the number of path segments intersected by rays is equal in counterclockwise and clockwise directions, the point is outside the shape. Otherwise, the point is inside the shape. You can understand this rule by using a counter, which starts with zero. Add one to the counter for every ray intersecting a path segment in the counterclockwise direction. Subtract one from the counter for every ray intersecting a path segment in the clockwise direction. At the end, if the counter is non-zero, the point is inside; otherwise, the point is outside. Figure 17-23 shows the same two paths

made of two triangular subpaths with their counter values when the `NON_ZERO` fill rule is applied. The rays drawn from the point are shown in dashed lines. The point in the first shape scores six (a non-zero value) and it is inside the path. The point in the second shape scores zero and it is outside the path.

Like the `NON_ZERO` fill rule, the `EVEN_ODD` fill rule also draws rays from a point in all directions extending to infinity, so all path segments are intersected. It counts the number of intersections between the rays and the path segments. If the number is odd, the point is inside the path. Otherwise, the point is outside the path. If you set the `fillRule` property to `EVEN_ODD` for the two shapes shown in Figure 17-23, the point is outside the path for both shapes because the number of intersections between rays and path segments is six (an even number) in both cases. The default value for the `fillRule` property of a `Path` is `FillRule.NON_ZERO`.



**Figure 17-23.** Applying the `NON_ZERO` fill rule to two triangular subpaths

The program in Listing 17-13 is an implementation of the examples discussed in this section. It draws four paths: the first two (counting from the left) with `NON_ZERO` fill rules and the last two with `EVEN_ODD` fill rules. Figure 17-24 shows the paths. The first and third paths use a counterclockwise stroke for drawing both triangular subpaths. The second and fourth paths are drawn using a counterclockwise stroke for one triangle and a clockwise stroke for another.

### **Listing 17-13.** Using Fill Rules for Paths

```
// PathFillRule.java
package com.jdojo.shape;

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

```
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.FillRule;
import javafx.scene.shape.LineTo;
import javafx.scene.shape.MoveTo;
import javafx.scene.shape.Path;
import javafx.scene.shape.PathElement;
import javafx.stage.Stage;

public class PathFillRule extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // Both triangles use a couterclockwise stroke
        PathElement[] pathEleemnts1 = {new MoveTo(50, 0),
            new LineTo(0, 50),
            new LineTo(100, 50),
            new LineTo(50, 0),
            new MoveTo(90, 15),
            new LineTo(40, 65),
            new LineTo(140, 65),
            new LineTo(90, 15)};

        // One traingle uses a clockwise stroke and
        // another uses a couterclockwise stroke
        PathElement[] pathEleemnts2 = {new MoveTo(50, 0),
            new LineTo(0, 50),
            new LineTo(100, 50),
            new LineTo(50, 0),
            new MoveTo(90, 15),
            new LineTo(140, 65),
            new LineTo(40, 65),
            new LineTo(90, 15)};

        /* Using the NON-ZERO fill rule by default */
        Path p1 = new Path(pathEleemnts1);
        p1.setFill(Color.LIGHTGRAY);

        Path p2 = new Path(pathEleemnts2);
        p2.setFill(Color.LIGHTGRAY);

        /* Using the EVEN_ODD fill rule */
        Path p3 = new Path(pathEleemnts1);
        p3.setFill(Color.LIGHTGRAY);
        p3.setFillRule(FillRule.EVEN_ODD);

        Path p4 = new Path(pathEleemnts2);
        p4.setFill(Color.LIGHTGRAY);
        p4.setFillRule(FillRule.EVEN_ODD);

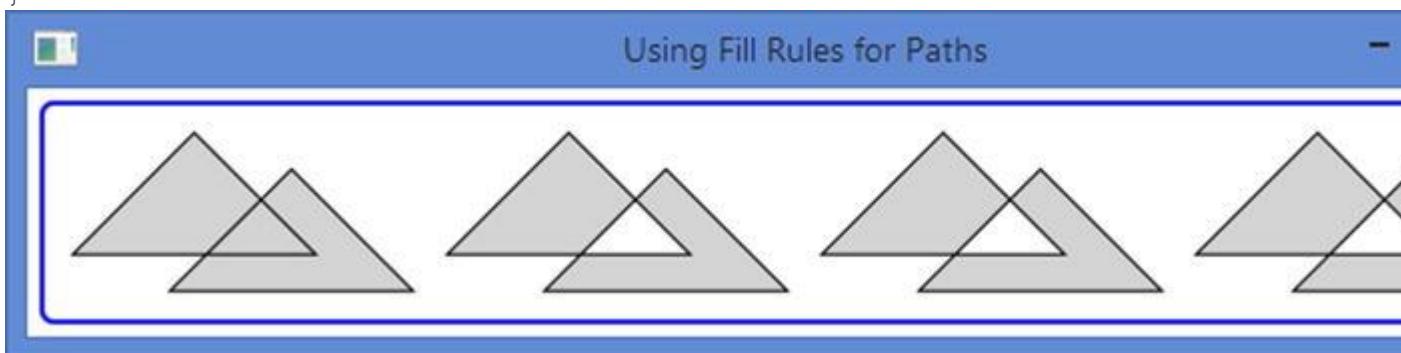
        HBox root = new HBox(p1, p2, p3, p4);
        root.setSpacing(10);
```

```

        root.setStyle("-fx-padding: 10;" +
                      "-fx-border-style: solid inside;" +
                      "-fx-border-width: 2;" +
                      "-fx-border-insets: 5;" +
                      "-fx-border-radius: 5;" +
                      "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using Fill Rules for Paths");
        stage.show();
    }
}

```



**Figure 17-24.** Paths using different fill rules

## Drawing Scalable Vector Graphics

An instance of the `SVGPath` class draws a shape from path data in an encoded string. You can find the SVG specification at <http://www.w3.org/TR/SVG>. You can find the detailed rules of constructing the path data in string format at <http://www.w3.org/TR/SVG/paths.html>. JavaFX partially supports SVG specification.

The `SVGPath` class contains a no-args constructor to create its object.

```
// Create a SVGPath object
SVGPath sp = new SVGPath();
```

The `SVGPath` class contains two properties.

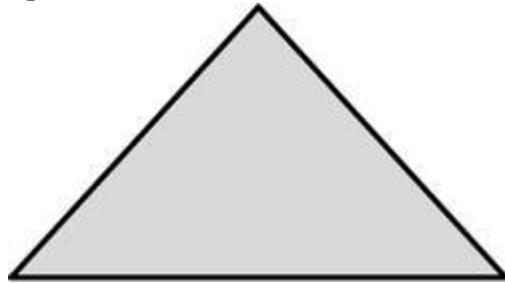
- `content`
- `fillRule`

The `content` property defines the encoded string for the SVG path. The `fillRule` property specifies the fill rule for the interior of the shape, which could be `FillRule.NON_ZERO` or `FillRule.EVEN_ODD`. The default value for the `fillRule` property is `FillRule.NON_ZERO`. Please refer to the

section “The Fill Rule for a Path” for more details on fill rules. Fill rule for a Path and a SVGPath work the same.

The following snippet of code sets “M50, o Lo, 50 L100, 50 Z” encoded string as the content for a SVGPath object to draw a triangle as shown in Figure 17-25:

```
SVGPath sp2 = new SVGPath();
sp2.setContent("M50, 0 L0, 50 L100, 50 Z");
sp2.setFill(Color.LIGHTGRAY);
sp2.setStroke(Color.BLACK);
```



**Figure 17-25.** A triangle using a SVGPath

The content of a SVGPath is an encoded string following some rules:

- The string consists of a series of commands.
- Each command name is exactly one-letter long.
- A command is followed by its parameters.
- Parameter values for a command are separated by a comma or a space. For example, “M50, o Lo, 50 L100, 50 Z” and “M50 o Lo 50 L100 50 Z” represent the same path. For readability, you will use a comma to separate two values.
- You do not need to add spaces before or after the command character. For example, “M50 o Lo 50 L100 50 Z” can be rewritten as “M50 oLo 50L100 50Z”.

Let us consider the SVG content used in the previous example.

M50, 0 L0, 50 L100, 50 Z

The content consists of four commands.

- M50, 0
- L0, 50
- L100, 50
- Z

Comparing the SVG path commands with the Path API, the first command is “MoveTo (50, 0)”; the second command is “LineTo(0, 50)”; the third command is “LineTo(100, 50)” and the fourth command is “ClosePath”.

**Tip** The command name in `SVGPath` content is the first letter of the classes representing path elements in a `Path` object. For example, an absolute `MoveTo` in the `Path` API becomes `M` in `SVGPathContent`, an absolute `LineTo` becomes `L`, and so on.

The parameters for the commands are coordinates, which can be absolute or relative. When the command name is in uppercase (e.g., `M`), its parameters are considered absolute. When the command name is in lowercase (e.g., `m`), its parameters are considered relative. The “closepath” command is `Z` or `z`. Because the “closepath” command does not take any parameters, both uppercase and lowercase versions behave the same.

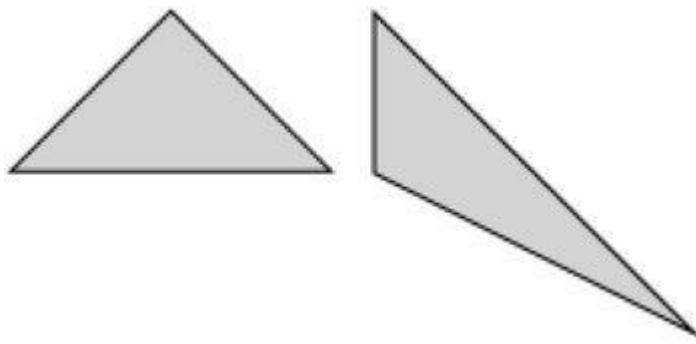
Consider the content of two SVG paths:

- `M50, 0 L0, 50 L100, 50 Z`
- `M50, 0 10, 50 1100, 50 Z`

The first path uses absolute coordinates. The second path uses absolute and relative coordinates. Like a `Path`, a `SVGPath` must start with a “moveTo” command, which must use absolute coordinates. If a `SVGPath` starts with a relative “moveTo” command (e.g., “`m 50, 0`”), its parameters are treated as absolute coordinates. In the foregoing SVG paths, you can start the string with “`m50, 0`” and the result will be the same.

The previous two SVG paths will draw two different triangles, as shown in Figure 17-26, even though both use the same parameters. The first path draws the triangle on the left and the second one draws the triangle on the right. The commands in the second path are interpreted as follows:

- Move to  $(50, 0)$
- Draw a line from the current point  $(50, 0)$  to  $(50, 50)$ . The ending point  $(50, 50)$  is derived by adding the x and y coordinates of the current point to the relative “lineto” command (`l`) parameters. The ending point becomes  $(50, 50)$ .
- Draw a line from the current point  $(50, 50)$  to  $(150, 100)$ . Again, the coordinates of the ending point are derived by adding the x and y coordinates of the current point  $(50, 50)$  to the command parameter “`l100, 50`” (The first character in “`l100, 50`” is the lowercase `L`, not the digit `1`).
- The close the path (`Z`)



**Figure 17-26.** Using absolute and relative coordinates in SVG paths

Table 17-2 lists the commands used in the content of the `SVGPath` objects. It also lists the equivalent classes used in the Path API. The table lists the command, which uses absolute coordinates. The relative versions of the commands use lowercase letters. The plus sign (+) in the parameter column indicates that multiple parameters may be used.

**Table 17-2.** List of SVG Path Commands

Command	Parameter	Command Name	Path API Class
M	(x, y)+	<code>moveto</code>	<code>MoveTo</code>
L	(x, y)+	<code>lineto</code>	<code>LineTo</code>
H	x+	<code>lineto</code>	<code>HLineTo</code>
V	y+	<code>lineto</code>	<code>VLineTo</code>
A	(rx, ry, x-axis-rotation, large-arc-flag, sweep-flag, x, y)+	<code>arcto</code>	<code>ArcTo</code>
Q	(x1, y1, x, y)+	<code>Quadratic Bezier curveto</code>	<code>QuadCurveTo</code>
T	(x, y)+	Shorthand/smooth quadratic Bezier curveto	<code>QuadCurveTo</code>
C	(x1, y1, x2, y2, x, y)+	<code>curveto</code>	<code>CubicCurveTo</code>
S	(x2, y2, x, y)+	Shorthand/smooth curveto	<code>CubicCurveTo</code>
Z	None	<code>closePath</code>	<code>ClosePath</code>

### The “moveTo” Command

The “moveTo” command `M` starts a new subpath at the specified (x, y) coordinates. It may be followed by one or multiple pairs of coordinates. The first pair of coordinates is considered the x and y coordinates of the point, which the command will make the current point. Each additional

pair is treated as a parameter for a “lineto” command. If the “moveTo” command is relative, the “lineto” command will be relative. If the “moveTo” command is absolute, the “lineto” command will be absolute. For example, the following two SVG paths are the same:

```
M50, 0 L0, 50 L100, 50 Z
M50, 0, 0, 50, 100, 50 Z
```

### The “lineto” Commands

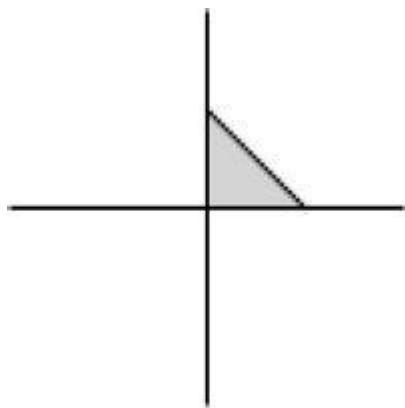
There are three “lineto” commands: L, H, and V. They are used to draw straight lines.

The command L is used to draw a straight line from the current point to the specified (x, y) point. If you specify multiple pairs of (x, y) coordinates, it draws a polyline. The final pair of the (x, y) coordinate becomes the new current point. The following SVG paths will draw the same triangle. The first one uses two L commands and the second one uses only one.

- M50, 0 L0, 50 L100, 50 L50, 0
- M50, 0 L0, 50, 100, 50, 50, 0

The H and V commands are used to draw horizontal and vertical lines from the current point. The command H draws a horizontal line from the current point (cx, cy) to (x, cy). The command V draws a vertical line from the current point (cx, cy) to (cx, y). You can pass multiple parameters to them. The final parameter value defines the current point. For example, “M0, 0H200, 100 V50Z” will draw a line from (0, 0) to (200, 0), from (200, 0) to (100, 0). The second command will make (100, 0) as the current point. The third command will draw a vertical line from (100, 0) to (100, 50). The z command will draw a line from (100, 50) to (0, 0). The following snippet of code draws a SVG path as shown in Figure 17-27:

```
SVGPath p1 = new SVGPath();
p1.setContent("M0, 0H-50, 50, 0 V-50, 50, 0, -25 L25, 0");
p1.setFill(Color.LIGHTGRAY);
p1.setStroke(Color.BLACK);
```



**Figure 17-27.** Using multiple parameters to “lineto” commands

### The “arcto” Command

The “arcto” command A draws an elliptical arc from the current point to the specified (x, y) point. It uses rx and ry as the radii along x-axis and y-axis. The x-axis-rotation is a rotation angle in degrees for the x-axis of the ellipse. The large-arc-flag and sweep-flag are the flags used to select one arc out of four possible arcs. Use 0 and 1 for flag values, where 1 means true and 0 means false. Please refer to the section “The ArcTo Path Element” for a detailed explanation of all its parameters. You can pass multiple arcs parameters, and in that case, the ending point of an arc becomes the current point for the subsequent arc. The following snippet of code draws two SVG paths with arcs. The first path uses one parameter for the “arcTo” command and the second path uses two parameters. Figure 17-28 shows the paths.

```
SVGPath p1 = new SVGPath();

// rx=150, ry=50, x-axis-rotation=0, large-arc-flag=0,
// sweep-flag 0, x=-50, y=50
p1.setContent("M0, 0 A150, 50, 0, 0, 0, -50, 50 Z");
p1.setFill(Color.LIGHTGRAY);
p1.setStroke(Color.BLACK);

// Use multiple arcs in one "arcTo" command
SVGPath p2 = new SVGPath();

// rx1=150, ry1=50, x-axis-rotation1=0, large-arc-flag1=0,
// sweep-flag1=0, x1=-50, y1=50
// rx2=150, ry2=10, x-axis-rotation2=0, large-arc-flag2=0,
// sweep-flag2=0, x2=10, y2=10
p2.setContent("M0, 0 A150 50 0 0 0 -50 50, 150 10 0 0 0 10 10
Z");
p2.setFill(Color.LIGHTGRAY);
p2.setStroke(Color.BLACK);
```



**Figure 17-28.** Using “*arcTo*” commands to draw elliptical arc paths

### The “Quadratic Bezier curveto” Command

Both commands `Q` and `T` are used to draw quadratic Bezier curve.

The command `Q` draws a quadratic Bezier curve from the current point to the specified  $(x, y)$  point using the specified  $(x_1, y_1)$  as the control point.

The command `T` draws a quadratic Bezier curve from the current point to the specified  $(x, y)$  point using a control point that is the reflection of the control point on the previous command. The current point is used as the control point if there was no previous command or the previous command was not `Q`, `q`, `T`, or `t`.

The command `Q` takes the control point as parameters whereas the command `T` assumes the control point. The following snippet of code uses the commands `Q` and `T` to draw quadratic Bezier curves as shown in Figure 17-29:

```
SVGPath p1 = new SVGPath();
p1.setContent("M0, 50 Q50, 0, 100, 50");
p1.setFill(Color.LIGHTGRAY);
p1.setStroke(Color.BLACK);

SVGPath p2 = new SVGPath();
p2.setContent("M0, 50 Q50, 0, 100, 50 T200, 50");
p2.setFill(Color.LIGHTGRAY);
p2.setStroke(Color.BLACK);
```



**Figure 17-29.** Using `Q` and `T` commands to draw quadratic Bezier curves

### The “Cubic Bezier curveto” Command

The commands `C` and `S` are used to draw cubic Bezier curves.

The command `C` draws a cubic Bezier curve from the current point to the specified point  $(x, y)$  using the specified controls points  $(x_1, y_1)$  and  $(x_2, y_2)$ .

The command `S` draws a cubic Bezier curve from the current point to the specified point  $(x, y)$ . It assumes the first control point to be the reflection of the second control point on the previous command. The current point is used as the first control point if there was no previous command or the previous command was not `C`, `c`, `S`, or `s`. The specified point  $(x_2, y_2)$  is the second control point. Multiple sets of coordinates draw a polybezier.

The following snippet of code uses the commands `C` and `S` to draw cubic Bezier curves as shown in Figure 17-30. The second path uses the command `S` to use the reflection of the second control point of the previous command `C` as its first control point.

```
SVGPath p1 = new SVGPath();
p1.setContent("M0, 0 C0, -100, 100, 100, 100, 0");
p1.setFill(Color.LIGHTGRAY);
p1.setStroke(Color.BLACK);

SVGPath p2 = new SVGPath();
p2.setContent("M0, 0 C0, -100, 100, 100, 100, 0 S200 100 200,
0");
p2.setFill(Color.LIGHTGRAY);
p2.setStroke(Color.BLACK);
```



**Figure 17-30.** Using `C` and `S` commands to draw cubic Bezier curves

### The “closepath” Command

The “closepath” commands `Z` and `z` draw a straight line from the current point to the starting point of the current subpath and ends the subpath. Both uppercase and lowercase versions of the command work the same.

### Combining Shapes

The `Shape` class provides three static methods that let you perform union, intersection and subtraction of shapes.

- `union(Shape shape1, Shape shape2)`
- `intersect(Shape shape1, Shape shape2)`
- `subtract(Shape shape1, Shape shape2)`

The methods return a new `Shape` instance. They operate on the areas of the input shapes. If a shape does not have a fill and a stroke, its area is zero. The new shape has a stroke and a fill.

The `union()` method combines the areas of two shapes.

The `intersect()` method uses the common areas between the shapes to create the new shape. The `subtract()` method creates a new shape by subtracting the specified second shape from the first shape.

The program in Listing 17-14 combines two circles using the `union`, `intersection`, and `subtraction` operations. Figure 17-31 shows the resulting shapes.

### ***Listing 17-14.*** Combining Shapes to Create New Shapes

```
// CombiningShapesTest.java
package com.jdojo.shape;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Shape;
import javafx.stage.Stage;

public class CombiningShapesTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Circle c1 = new Circle (0, 0, 20);
        Circle c2 = new Circle (15, 0, 20);

        Shape union = Shape.union(c1, c2);
        union.setStroke(Color.BLACK);
        union.setFill(Color.LIGHTGRAY);

        Shape intersection = Shape.intersect(c1, c2);
        intersection.setStroke(Color.BLACK);
        intersection.setFill(Color.LIGHTGRAY);

        Shape subtraction = Shape.subtract(c1, c2);
        subtraction.setStroke(Color.BLACK);
        subtraction.setFill(Color.LIGHTGRAY);

        HBox root = new HBox(union, intersection,
subtraction);
        root.setSpacing(20);
        root.setStyle("-fx-padding: 10;" +
"-fx-border-style: solid inside;" +
"-fx-border-width: 2;" +
"-fx-border-insets: 5;" +
"-fx-border-radius: 5;" +
"-fx-border-color: blue;");
```



**Figure 17-31.** Shapes created by combining two circles

## Understanding the Stroke of a Shape

Stroking is the process of painting the outline of a shape. Sometimes, the outline of a shape is also known as stroke. The `Shape` class contains several properties to define the appearance of the stroke of a shape.

- `stroke`
- `strokeWidth`
- `strokeType`
- `strokeLineCap`
- `strokeLineJoin`
- `strokeMiterLimit`
- `strokeDashOffset`

The `stroke` property specifies the color of the stroke. The default `stroke` is set to `null` for all shapes except `Line`, `Path` and `Polyline`, which have `Color.BLACK` as their default stroke.

The `strokeWidth` property specifies the width of the stroke. It is `1.0px` by default.

The stroke is painted along the boundary of a shape. The `strokeType` property specifies the distribution of the width of the stroke on the boundary. Its value is one of the three constants, `CENTERED`, `INSIDE`, and `OUTSIDE`, the `StrokeType` enum. The default value is `CENTERED`. The `CENTERED` stroke type draws a half of the stroke width outside and half inside the boundary. The `INSIDE` stroke type draws the stroke inside the boundary.

The OUTSIDE stroke draws the stroke outside the boundary. The stroke width of a shape is included in its layout bounds.

The program in Listing 17-15 creates four rectangles as shown in Figure 17-32. All rectangles have the same width and height (50px and 50px). The first rectangle, counting from the left, has no stroke and it has layout bounds of 50px X 50px. The second rectangle uses a stroke of width 4px and an INSIDE stroke type. The INSIDE stroke type is drawn inside the width and height boundary, the rectangle has the layout bounds of 50px X 50px. The third rectangle uses a stroke width 4px and a CENTERED stroke type, which is the default. The stroke is drawn 2px inside the boundary and 2px outside the boundary. The 2px outside stroke is added to the dimensions of all four making the layout bounds to 54px X 54px. The fourth rectangle uses a 4px stroke width and an OUTSIDE stroke type. The entire stroke width falls outside the width and height of the rectangle making the layouts to 58px X 58px.

### ***Listing 17-15.*** Effects of Applying Different Stroke Types on a Rectangle

```
// StrokeTypeTest.java
package com.jdojo.shape;

import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.shape.StrokeType;
import javafx.stage.Stage;

public class StrokeTypeTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Rectangle r1 = new Rectangle(50, 50);
        r1.setFill(Color.LIGHTGRAY);

        Rectangle r2 = new Rectangle(50, 50);
        r2.setFill(Color.LIGHTGRAY);
        r2.setStroke(Color.BLACK);
        r2.setStrokeWidth(4);
        r2.setStrokeType(StrokeType.INSIDE);

        Rectangle r3 = new Rectangle(50, 50);
        r3.setFill(Color.LIGHTGRAY);
        r3.setStroke(Color.BLACK);
```

```

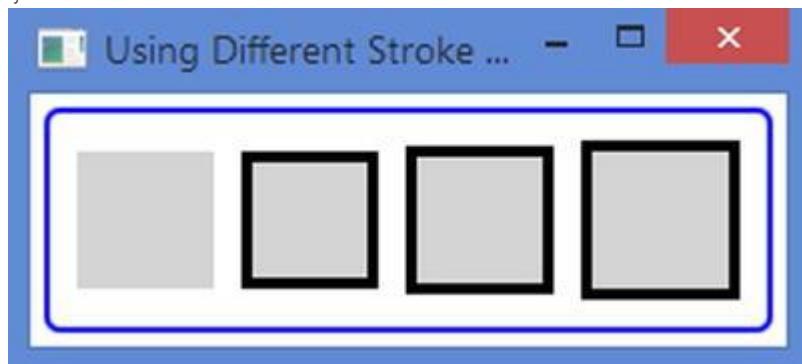
r3.setStrokeWidth(4);

Rectangle r4 = new Rectangle(50, 50);
r4.setFill(Color.LIGHTGRAY);
r4.setStroke(Color.BLACK);
r4.setStrokeWidth(4);
r4.setStrokeType(StrokeType.OUTSIDE);

HBox root = new HBox(r1, r2, r3, r4);
root.setAlignment(Pos.CENTER);
root.setSpacing(10);
root.setStyle("-fx-padding: 10;" +
    "-fx-border-style: solid inside;" +
    "-fx-border-width: 2;" +
    "-fx-border-insets: 5;" +
    "-fx-border-radius: 5;" +
    "-fx-border-color: blue;");

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("Using Different Stroke Types for
Shapes");
stage.show();
}
}

```



**Figure 17-32.** Rectangles using different types of strokes

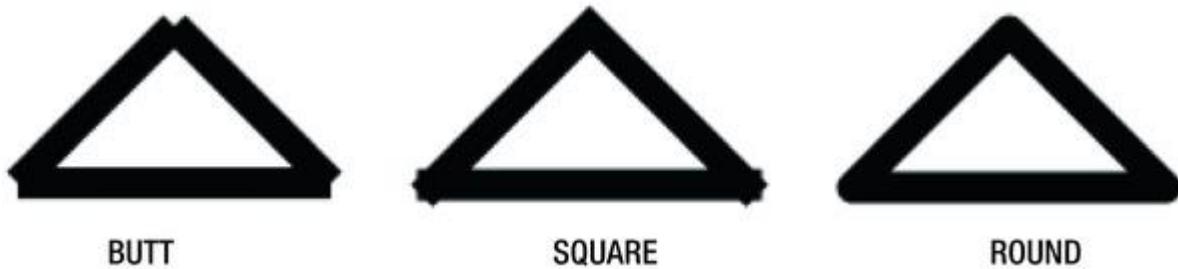
The `strokeLineCap` property specifies the ending decoration of a stroke for unclosed subpaths and dash segments. Its value is one of the constants of the `StrokeLineCap` enum: `BUTT`, `SQUARE`, and `ROUND`. The default is `BUTT`. The `BUTT` line cap adds no decoration to the end of a subpath; the stroke starts and ends exactly at the starting and ending points. The `SQUARE` line cap extends the end by half the stroke width. The `ROUND` line cap adds a round cap to the end. The round cap uses a radius equal to half the stroke width. Figure 17-33 shows three lines, which are unclosed subpaths. All lines are 100px wide using 10px stroke width. The figure shows the `strokeLineCap` they use. The width of the layout bounds of the line using the `BUTT` line cap remains 100px.

However, for other two lines, the width of the layout bounds increases to 110px—increasing by 10px at both ends.



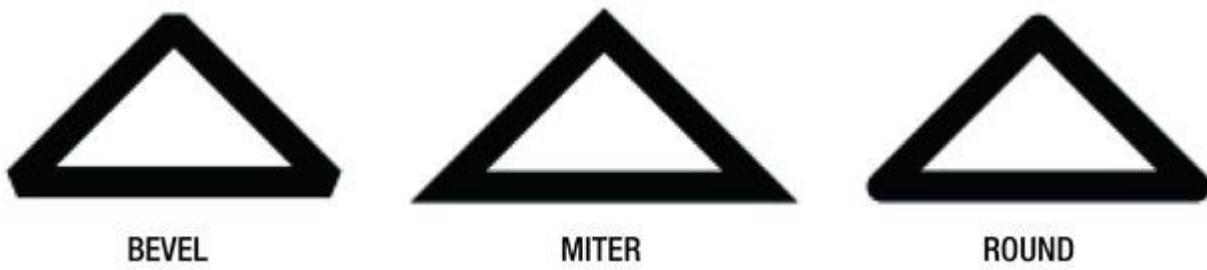
**Figure 17-33.** Different line cap style for strokes

Note that the `strokeLineCap` properties are applied to the ends of a line segment of `unclosedSubpaths`. Figure 17-34 shows three triangles created by unclosed subpaths. They use different stroke line caps. The SVG path data “M50, oLo, 50 Mo, 50 L100, 50 M100, 50 L50, o” was used to draw the triangles. The fill was set to `null` and the stroke width to 10px.



**Figure 17-34.** Triangles using unclosed subpaths using different stroke line caps

The `strokeLineJoin` property specifies how two successive path elements of a subpath are joined. Its value is one of the constants of the `StrokeLineJoin` enum: `BEVEL`, `MITER`, and `ROUND`. The default is `MITER`. The `BEVEL` line join connects the outer corners of path elements by a straight line. The `MITER` line join extends the outer edges of two path elements until they meet. The `ROUND` line join connects two path elements by rounding their corners by half the stroke width. Figure 17-35 shows three triangles created with the SVG path data “M50, oLo, 50 L100, 50 Z”. The fill color is `null` and the stroke width is 10px. The triangles use different line joins as shown in the figure.



**Figure 17-35.** Triangles using different stroke line join types

A MITER line join joins two path elements by extending their outer edges. If the path elements meet at a smaller angle, the length of the join may become very big. You can limit the length of the join using the `strokeMiterLimit` property. It specifies the ratio of the miter length and the stroke width. The miter length is the distance between the most inside point and the most outside point of the join. If the two path elements cannot meet by extending their outer edges within this limit, a BEVEL join is used instead. The default value is 10.0. That is, by default, the miter length may be up to ten times the stroke width.

The following snippet of code creates two triangles as shown in Figure 17-36. Both use MITER line join by default. The first triangle uses 2.0 as the miter limit. The second triangle uses the default miter limit, which is 10.0. The stroke width is 10px. The first triangle tries to join the corners by extending two lines up to 20px, which is computed by multiplying the 10px stroke width by the miter limit of 2.0. The corners cannot be joined using the MITER join within 20px, so a BEVEL join is used.

```
SVGPath t1 = new SVGPath();
t1.setContent("M50, 0L0, 50 L100, 50 Z");
t1.setStrokeWidth(10);
t1.setFill(null);
t1.setStroke(Color.BLACK);
t1.setStrokeMiterLimit(2.0);

SVGPath t2 = new SVGPath();
t2.setContent("M50, 0L0, 50 L100, 50 Z");
t2.setStrokeWidth(10);
t2.setFill(null);
t2.setStroke(Color.BLACK);
```



**Figure 17-36.** Triangles using different stroke miter limits

By default, the stroke draws a solid outline. You can also have a dashed outline. You need to provide a dashing pattern and a dash offset. The dashing pattern is an `array of double` that is stored in an `ObservableList<Double>`. You can get the reference of the list using the `getStrokeDashArray()` method of the `Shape` class. The elements of the list specify a pattern of dashes and gaps. The first element is the dash length, the second gap, the third dash length, the fourth gap, and so on. The dashing pattern is repeated to draw the

outline. The `strokeDashOffset` property specifies the offset in the dashing pattern where the stroke begins.

The following snippet of code creates two instances of `Polygon` as shown in Figure 17-37. Both use the same dashing patterns but a different dash offset. The first one uses the dash offset of 0.0, which is the default. The stroke of the first rectangle starts with a 15.0px dash, which is the first element of the dashing pattern, which can be seen in the dashed line drawn from the (0, 0) to (100, 0). The second `Polygon` uses a dash offset of 20.0, which means the stroke will start 20.0px inside the dashing pattern. The first two elements 15.0 and 3.0 are inside the dash offset 20.0. Therefore, the stroke for the second `Polygon` starts at the third element, which is a 5.0px dash.

```
Polygon p1 = new Polygon(0, 0, 100, 0, 100, 50, 0, 50, 0, 0);
p1.setFill(null);
p1.setStroke(Color.BLACK);
p1.getStrokeDashArray().addAll(15.0, 5.0, 5.0, 5.0);

Polygon p2 = new Polygon(0, 0, 100, 0, 100, 50, 0, 50, 0, 0);
p2.setFill(null);
p2.setStroke(Color.BLACK);
p2.getStrokeDashArray().addAll(15.0, 5.0, 5.0, 5.0);
p2.setStrokeDashOffset(20.0);
```



**Figure 17-37.** Two polygons using dashing patterns for their outline

## Styling Shapes with CSS

All shapes do not have a default style-class name. If you want to apply styles to shapes using CSS, you need to add style-class names to them. All shapes can use the following CSS properties:

- `-fx-fill`
- `-fx-smooth`
- `-fx-stroke`
- `-fx-stroke-type`
- `-fx-stroke-dash-array`
- `-fx-stroke-dash-offset`
- `-fx-stroke-line-cap`
- `-fx-stroke-line-join`
- `-fx-stroke-miter-limit`
- `-fx-stroke-width`

All CSS properties correspond to the properties in the `Shape` class, which I have discussed at length in the previous section. `Rectangle` supports two additional CSS properties to specify arc width and height for rounded rectangles:

- `-fx-arc-height`
- `-fx-arc-width`

The following snippet of code creates a `Rectangle` and adds `rectangle` as its style-class name:

```
Rectangle r1 = new Rectangle(200, 50);
r1.getStyleClass().add("rectangle");
```

The following style will produce a rectangle as shown in Figure 17-38:

```
.rectangle {
    -fx-fill: lightgray;
    -fx-stroke: black;
    -fx-stroke-width: 4;
    -fx-stroke-dash-array: 15 5 5 10;
    -fx-stroke-dash-offset: 20;
    -fx-stroke-line-cap: round;
    -fx-stroke-line-join: bevel;
}
```



**Figure 17-38.** Applying CSS styles to a rectangle

## Summary

Any shape that can be drawn in a two-dimensional plane is called a 2D shape. JavaFX offers various nodes to draw different types of shapes (lines, circles, rectangles, etc.). You can add shapes to a scene graph. All shape classes are in the `javafx.scene.shape` package. Classes representing 2D shapes are inherited from the abstract `Shape` class. A shape can have a stroke that defines the outline of the shape. A shape may have a fill.

An instance of the `Line` class represents a line node. A `Line` has no interior. By default, its `fill` property is set to `null`. Setting `fill` has no effect. The default `stroke` is `Color.BLACK` and the default `strokeWidth` is `1.0`.

An instance of the `Rectangle` class represents a rectangle node. The class uses six properties to define the rectangle: `x`, `y`, `width`, `height`, `arcWidth`, and `arcHeight`.

The `x` and `y` properties are the `x` and `y` coordinates of the upper-left corner of the rectangle in the local coordinate system of the node. The `width` and `height` properties are the width and height of the rectangle, respectively. Specify the same width and height to draw a square. By default, the corners of a rectangle are sharp. A rectangle can have rounded corners by specifying the `arcWidth` and `arcHeight` properties.

An instance of the `Circle` class represents a circle node. The class uses three properties to define the circle: `centerX`, `centerY`, and `radius`. The `centerX` and `centerY` properties are the `x` and `y` coordinates of the center of the circle in the local coordinate system of the node. The `radius` property is the radius of the circle. The default values for these properties are zero.

An instance of the `Ellipse` class represents an ellipse node. The class uses four properties to define the ellipse: `centerX`, `centerY`, `radiusX`, `radiusY`. The `centerX` and `centerY` properties are the `x` and `y` coordinates of the center of the circle in the local coordinate system of the node. The `radiusX` and `radiusY` are the radii of the ellipse in the horizontal and vertical directions. The default values for these properties are zero. A circle is a special case of an ellipse when `radiusX` and `radiusY` are the same.

An instance of the `Polygon` class represents a polygon node. The class does not define any public properties. It lets you draw a polygon using an array of (`x`, `y`) coordinates defining the vertices of the polygon. Using the `Polygon` class, you can draw any type of geometric shape that is created using connected lines (triangles, pentagon, hexagon, parallelogram, etc.).

A polyline is similar to a polygon, except that it does not draw a line between the last and first points. That is, a polyline is an open polygon. However, the `fill` color is used to fill the entire shape as if the shape was closed. An instance of the `Polyline` class represents a polyline node.

An instance of the `Arc` class represents a sector of an ellipse. The class uses seven properties to define the ellipse: `centerX`, `centerY`, `radiusX`, `radiusY`, `startAngle`, `length`, and `type`. The first four properties define an ellipse. The last three properties define a sector of the ellipse that is the `Arc` node. The `startAngle` property specifies the start angle of the section in degrees measured counterclockwise from the positive `x`-axis. It defines the beginning of the arc. The `length` is an angle in degrees measured

counterclockwise from the start angle to define the end of the sector. If the `length` property is set to 360, the `Arc` is a full ellipse.

Bezier curves are used in computer graphics to draw smooth curves. An instance of the `QuadCurve` class represents a quadratic Bezier curve segment intersecting two specified points using a specified Bezier control point.

An instance of the `CubicCurve` class represents a cubic Bezier curve segment intersecting two specified points using two specified Bezier control points.

You can draw complex shapes using the `Path` class. An instance of the `Path` class defines the path (outline) of a shape. A path consists of one or more subpaths. A subpath consists of one or more path elements. Each subpath has a starting point and an ending point. A path element is an instance of the `PathElement` abstract class. Several subclasses of the `PathElement` class exist to represent specific type of path elements; those classes are `MoveTo`, `LineTo`, `HLineTo`, `VLineTo`, `ArcTo`, `QuadCurveTo`, `CubicCurveTo`, and `ClosePath`.

JavaFX partially supports SVG specification. An instance of the `SVGPath` class draws a shape from path data in an encoded string.

JavaFX lets you create a shape by combining multiple shapes.

The `Shape` class provides three static methods

named `union()`, `intersect()`, and `subtract()` that let you perform union, intersection, and subtraction of two shapes that are passed as the arguments to these methods. The methods return a new `Shape` instance. They operate on the areas of the input shapes. If a shape does not have a fill and a stroke, its area is zero. The new shape has a stroke and a fill. The `union()` method combines the areas of two shapes.

The `intersect()` method uses the common areas between the shapes to create the new shape. The `subtract()` method creates a new shape by subtracting the specified second shape from the first shape.

Stroking is the process of painting the outline of a shape. Sometimes, the outline of a shape is also known as stroke. The `Shape` class contains several properties such as `stroke`, `strokeWidth`, and so on to define the appearance of the stroke of a shape.

JavaFX lets you style 2D shapes with CSS.

## CHAPTER 18



### Understanding Text Nodes

In this chapter, you will learn:

- What a `Text` node is and how to create it
- The coordinate system used for drawing `Text` nodes
- How to display multiline text in a `Text` node
- How to set fonts for a `Text` node
- How to access installed fonts and how to install custom fonts
- How to set the fill and stroke for `Text` nodes
- How to apply decoration such as underline and strikethrough to `Text` nodes
- How to apply font smoothing
- How to style `Text` nodes using CSS

#### What Is a Text Node?

A text node is an instance of the `Text` class that is used to render text. The `Text` class contains several properties to customize the appearance of text. The `Text` class and all its related classes – for example, the `Font` class, the `TextAlignment` enum, the `FontWeight` enum, etc. – are in the `javafx.scene.text` package.

The `Text` class inherits from the `Shape` class. That is, a `Text` is a `Shape`, which allows you to use all properties and methods of the `Shape` class on a `Text` node. For example, you can apply a fill color and a stroke to a `Text` node. Because `Text` is a node, you can use features of the `Node` class: for example, applying effects and transformations. You can also set text alignment, font family, font size, text wrapping style, etc., on a `Text` node.

Figure 18-1 shows three text nodes. The first one (from the left) is a simple text node. The second one uses bold text in a bigger font size. The third one uses the `Reflection` effect, a bigger font size, a stroke, and a fill.



**Figure 18-1.** A window showing three `Text` nodes

## Creating a Text Node

An instance of the `Text` class represents a `Text` node. A `Text` node contains text and properties to render the text. You can create a `Text` node using one of the constructors of the `Text` class:

- `Text()`
- `Text(String text)`
- `Text(double x, double y, String text)`

The no-args constructor creates a `Text` node with an empty string as its text. Other constructors let you specify the text and position the node.

The `text` property of the `Text` class specifies the text (or content) of the `Text` node. The `x` and `y` properties specify the `x` and `y` coordinates of the text origin, which are described in the next section.

```
// Create an empty Text Node and later set its text
Text t1 = new Text();
t1.setText("Hello from the Text node!");

// Create a Text Node with initial text
Text t2 = new Text("Hello from the Text node!");

// Create a Text Node with initial text and position
Text t3 = new Text(50, 50, "Hello from the Text node!");
```

**Tip** The width and height of a text node are automatically determined by its font. By default, a `Text` node uses a system default font to render its text.

The program in Listing 18-1 creates three `Text` nodes, sets their different properties, and adds them to an `HBox`. The `Text` nodes are displayed as shown in Figure 18-1.

### ***Listing 18-1.*** Creating Text Nodes

```
// TextTest.java
package com.jdojo.shape;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.effect.Reflection;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class TextTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
```

```

    }

@Override
public void start(Stage stage) {
    Text t1 = new Text("Hello Text Node!");

    Text t2 = new Text("Bold and Big");
    t2.setFont(Font.font("Tahoma", FontWeight.BOLD,
16));

    Text t3 = new Text("Reflection");
    t3.setEffect(new Reflection());
    t3.setStroke(Color.BLACK);
    t3.setFill(Color.WHITE);
    t3.setFont(Font.font("Arial", FontWeight.BOLD, 20));

    HBox root = new HBox(t1, t2, t3);
    root.setSpacing(20);
    root.setStyle("-fx-padding: 10;" +
                  "-fx-border-style: solid inside;" +
                  "-fx-border-width: 2;" +
                  "-fx-border-insets: 5;" +
                  "-fx-border-radius: 5;" +
                  "-fx-border-color: blue;");

    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("Using Text Nodes");
    stage.show();
}
}

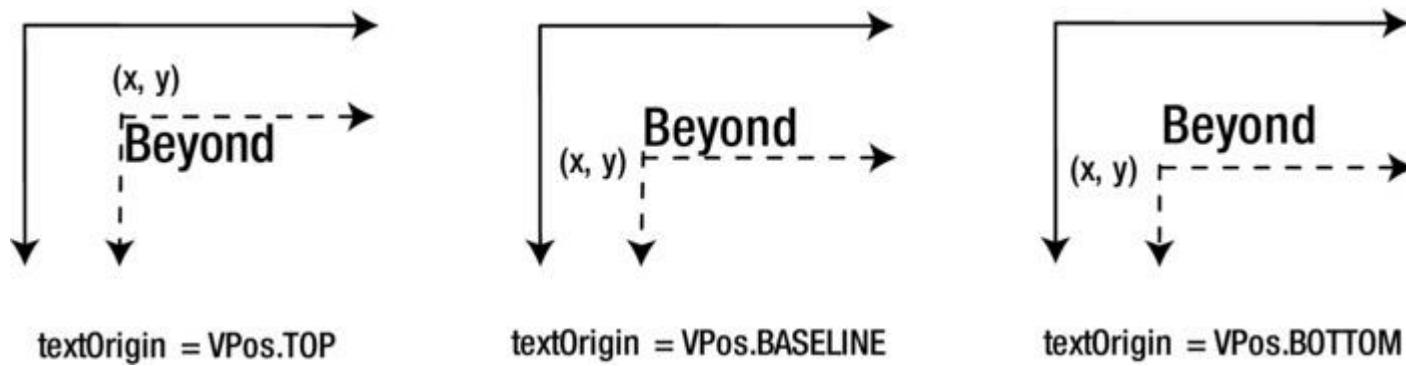
```

## Understanding the Text Origin

Apart from the local and parent coordinate system, a `Text` node has an additional coordinate system. It is the coordinate system used for drawing the text. Three properties of the `Text` class define the text coordinate system:

- `x`
- `y`
- `textOrigin`

The `x` and `y` properties define the `x` and `y` coordinates of the text origin. The `textOrigin` property is of type `VPos`. Its value could be `VPos.BASELINE`, `VPos.TOP`, `VPos.CENTER`, and `VPos.BOTTOM`. The default is `VPos.BASELINE`. It defines where the `x`-axis of the text coordinate system lies within the text height. Figure 18-2 shows the local and text coordinate systems of a text node. The local coordinate axes are in solid lines. The text coordinate axes are in dashed lines.



**Figure 18-2.** Effects of the `textOrigin` property on the vertical location of text drawing

When the `textOrigin` is `VPos.TOP`, the x-axis of the text coordinate system is aligned with the top of the text. That is, the `y` property of the `Text` node is the distance between the x-axis of the local coordinate system and the top of the displayed text. A font places its characters on a line called the *baseline*. The `VPos.BASELINE` aligns the x-axis of the text coordinate system with the baseline of the font. Note that some characters (e.g., g, y, j, p, etc.) are extended below the baseline. The `VPos.BOTTOM` aligns the x-axis of the text coordinate system with the bottom of the displayed text accounting for the descent for the font. The `VPos.CENTER` (not shown in the figure) aligns the x-axis of the text coordinate system in the middle of the displayed text, accounting for the ascent and descent for the font.

**Tip** The `Text` class contains a read-only `baselineOffset` property. Its value is the vertical distance between the top and baseline of the text. It is equal to the max ascent of the font.

Most of the time, you need not worry about the `textOrigin` property of the `Text` node, except when you need to align it vertically relative to another node. Listing 18-2 shows how to center a `Text` node horizontally and vertically in a scene. To center the node vertically, you must set the `textOrigin` property to `VPos.TOP`. The text is displayed as shown in Figure 18-3. If you do not set the `textOrigin` property, its y-axis is aligned with its baseline and it appears above the centerline of the scene.

### ***Listing 18-2.*** Centering a Text Node in a Scene

```
// TextCentering.java
package com.jdojo.shape;

import javafx.application.Application;
import javafx.geometry.VPos;
import javafx.scene.Group;
```

```

import javafx.scene.Scene;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class TextCentering extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

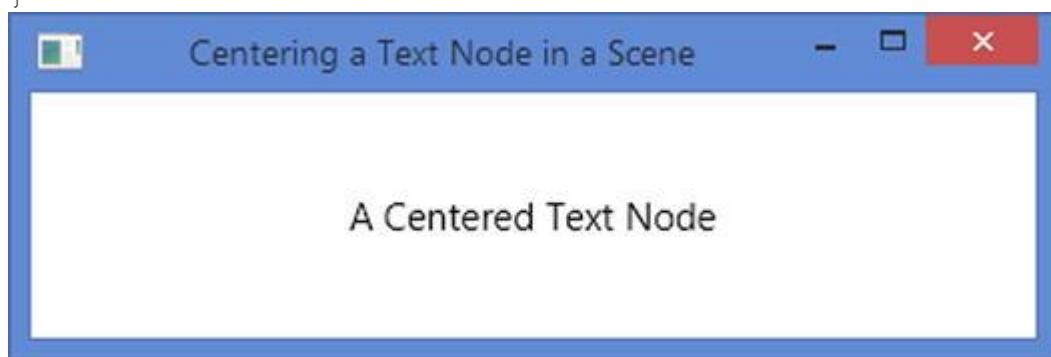
    @Override
    public void start(Stage stage) {
        Text msg = new Text("A Centered Text Node");

        // Must set the textOrigin to VPos.TOP to center
        // the text node vertically within the scene
        msg.setTextOrigin(VPos.TOP);

        Group root = new Group();
        root.getChildren().addAll(msg);
        Scene scene = new Scene(root, 200, 50);
        msg.layoutXProperty().bind(scene.widthProperty().subtract(
            msg.layoutBoundsProperty().get().getWidth()).divide(2));
        msg.layoutYProperty().bind(scene.heightProperty().subtract(
            msg.layoutBoundsProperty().get().getHeight()).divide(2));

        stage.setTitle("Centering a Text Node in a Scene");
        stage.setScene(scene);
        stage.sizeToScene();
        stage.show();
    }
}

```



**Figure 18-3.** A Text node centered in a scene

## Displaying Multiline Text

A Text node is capable of displaying multiple lines of text. It creates a new line in two cases:

- A newline character ‘\n’ in the text creates a new line causing the characters following the newline to wrap to the next line.
  - The `Text` class contains a `wrappingWidth` property, which is 0.0 by default. Its value is specified in pixels, not characters. If it is greater than zero, the text in each line is wrapped to at the specified value.

The `lineSpacing` property specifies the vertical spacing in pixels between two lines. It is 0.0 by default.

The `textAlignment` property specifies the horizontal alignment of the text lines in the bounding box. The widest line defines the width of the bounding box. Its value has no effect in a single line `Textnode`. Its value can be one of the constants of the `TextAlignment` enum: `LEFT`, `RIGHT`, `CENTER`, and `JUSTIFY`. The default is `TextAlignment.LEFT`.

The program in Listing 18-3 creates three multiline Text nodes as shown in Figure 18-4. The text for all nodes is the same. The text contains three newline characters. The first node uses the default LEFTtext alignment and a line spacing of 5px. The second node uses RIGHT text alignment with the default line spacing of opx. The third node uses a wrappingWidth of 100px. A new line is created at 100px as well as a newline character '\n'.

### ***Listing 18-3.*** Using Multiline Text Nodes

```
// MultilineText.java
package com.jdojo.shape;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.text.Text;
import javafx.scene.text.TextAlignment;
import javafx.stage.Stage;

public class MultilineText extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        String text = "Strange fits of passion have I known:  
\\n" +
                     "And I will dare to tell, \\n" +
                     "But in the lover's ear alone, \\n" +
                     "What once to me befell.";
```

```

Text t1 = new Text(text);
t1.setLineSpacing(5);

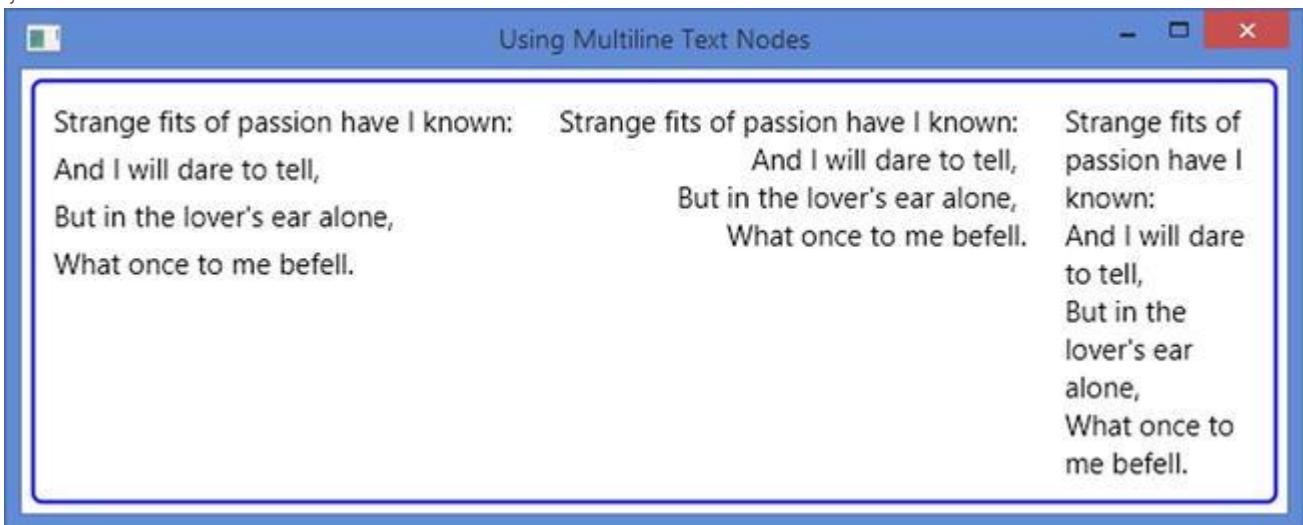
Text t2 = new Text(text);
t2.setTextAlignment(TextAlignment.RIGHT);

Text t3 = new Text(text);
t3.setWrappingWidth(100);

HBox root = new HBox(t1, t2, t3);
root.setSpacing(20);
root.setStyle("-fx-padding: 10;" +
              "-fx-border-style: solid inside;" +
              "-fx-border-width: 2;" +
              "-fx-border-insets: 5;" +
              "-fx-border-radius: 5;" +
              "-fx-border-color: blue;");

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("Using Multiline Text Nodes");
stage.show();
}
}

```



**Figure 18-4.** Multiline Text nodes

## Setting Text Fonts

The `font` property of the `Text` class defines the font for the text. The default font used is from “System” font family with the “Regular” style. The size of the default font is dependent on the platform and the desktop settings of the user.

A font has a *family* and a *family name*. A font family is also known as a *typeface*. A font family defines shapes (or glyphs) for characters. The same characters appear differently when displayed using fonts belonging

to different font families. Variants of a font are created by applying styles. Each variant of the font has a name that consists of the family name and the style names. For example, “Arial” is a family name of a font whereas “Arial Regular,” “Arial Bold,” and “Arial Bold Italic” are names of the variants of the “Arial” font.

## Creating Fonts

An instance of the `Font` class represents a font. The `Font` class provides two constructors:

- `Font(double size)`
- `Font(String name, double size)`

The first constructor creates a `Font` object of the specified size that belongs to the “System” font family. The second one creates a `Font` object of the specified full name of the font and the specified size. The size of the font is specified in points. The following snippet of code creates some font objects of the “Arial” family.

The `getFamily()`, `getName()`, and `getSize()` methods of the `Font` class return the family name, full name, and size of the font, respectively.

```
// Arial Plain  
Font f1 = new Font("Arial", 10);  
  
// Arial Italic  
Font f2 = new Font("Arial Italic", 10);  
  
// Arial Bold Italic  
Font f3 = new Font("Arial Bold Italic", 10);  
  
// Arial Narrow Bold  
Font f4 = new Font("Arial Narrow Bold", 30);
```

If the full font name is not found, the default “System” font will be created. It is hard to remember or know the full names for all variants of a font. To address this, the `Font` class provides factory methods to create fonts using a font family name, styles, and size:

- `font(double size)`
- `font(String family)`
- `font(String family, double size)`
- `font(String family, FontPosture posture, double size)`
- `font(String family, FontWeight weight, double size)`

- `font(String family, FontWeight weight, FontPosture posture, double size)`

The `font()` methods let you specify the family name, font weight, font posture, and font size. If only the family name is provided, the default font size is used, which depends on the platform and the desktop setting of the user.

The font weight specifies how bold the font is. Its value is one of the constants of the `FontWeight` enum: THIN, EXTRA\_LIGHT, LIGHT, NORMAL, MEDIUM, SEMI\_BOLD, BOLD, EXTRA\_BOLD, BLACK. The constant THIN represents the thinnest font and the constant BLACK the thickest font.

The posture of a font specifies whether it is italicized. It is represented by one of the two constants of

the `FontPosture` enum: REGULAR and ITALIC.

The following snippet of code creates fonts using the factory methods of the `Font` class.

```
// Arial Regular
Font f1 = Font.font("Arial", 10);

// Arial Bold
Font f2 = Font.font("Arial", FontWeight.BOLD, 10);

// Arial Bold Italic
Font f3 = Font.font("Arial", FontWeight.BOLD, FontPosture.ITALIC,
10);

// Arial THIN
Font f4 = Font.font("Arial", FontWeight.THIN, 30);
```

**Tip** Use the `getDefault()` static method of the `Font` class to get the system default font.

The program in Listing 18-4 creates `Text` nodes and sets their `font` property. The first `Text` node uses the default font. Figure 18-5 shows the `Text` nodes. The text for the `Text` nodes is the `String` returned from the `toString()` method of their `Font` objects.

#### ***Listing 18-4.*** Setting Fonts for Text Nodes

```
// TextFontTest.java
package com.jdojo.shape;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
```

```
import javafx.scene.text.Font;
import javafx.scene.text.FontPosture;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class TextFontTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Text t1 = new Text();
        t1.setText(t1.getFont().toString());

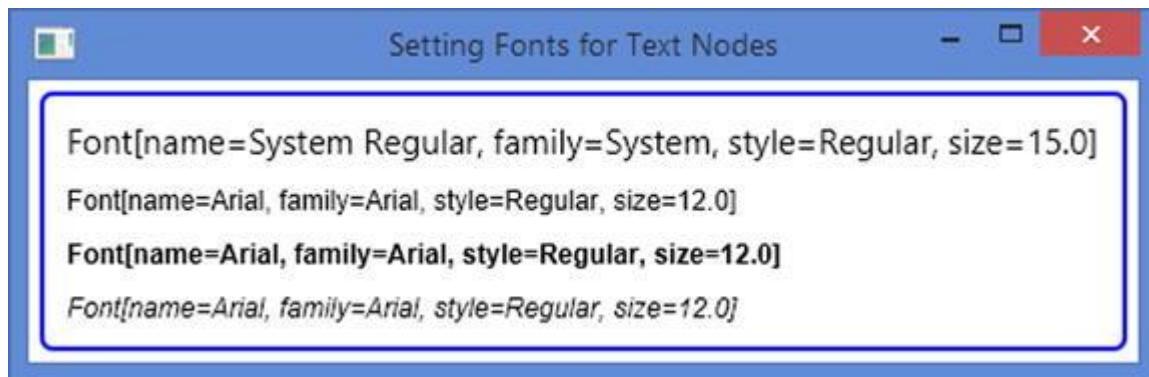
        Text t2 = new Text();
        t2.setFont(Font.font("Arial", 12));
        t2.setText(t2.getFont().toString());

        Text t3 = new Text();
        t3.setFont(Font.font("Arial", FontWeight.BLACK,
12));
        t3.setText(t2.getFont().toString());

        Text t4 = new Text();
        t4.setFont(Font.font("Arial", FontWeight.THIN,
FontPosture.ITALIC, 12));
        t4.setText(t2.getFont().toString());

        VBox root = new VBox(t1, t2, t3, t4);
        root.setSpacing(10);
        root.setStyle("-fx-padding: 10;" +
"-fx-border-style: solid inside;" +
"-fx-border-width: 2;" +
"-fx-border-insets: 5;" +
"-fx-border-radius: 5;" +
"-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Setting Fonts for Text Nodes");
        stage.show();
    }
}
```



**Figure 18-5.** Text nodes using variants of the “Arial” font family

### Accessing Installed Fonts

You can get the list of installed fonts on your machine. You can get the list of font family names, full font names, and full font names for a specified family name for all installed fonts. The following static methods in the `Font` class provide these lists.

- `List<String> getFamilies()`
- `List<String> getFontNames()`
- `List<String> getFontNames(String family)`

The following snippet of code prints the family names of all installed fonts on a machine. The output was generated on Windows. A partial output is shown:

```
// Print the family names of all installed fonts
for(String familyName: Font.getFamilies()) {
    System.out.println(familyName);
}
Agency FB
Algerian
Arial
Arial Black
Arial Narrow
Arial Rounded MT Bold
...
```

The following snippet of code prints the full names of all installed fonts on a machine. The output was generated on Windows. A partial output is shown:

```
// Print the full names of all installed fonts
for(String fullName: Font.getFontNames()) {
    System.out.println(fullName);
}
Agency FB
Agency FB Bold
Algerian
```

```
Arial
Arial Black
Arial Bold
Arial Bold Italic
Arial Italic
Arial Narrow
Arial Narrow Bold
Arial Narrow Bold Italic
More output goes here...
```

The following snippet of code prints the full names of all installed fonts for the “Times New Roman” family:

```
// Print the full names of "Times New Roman" family
for(String fullName: Font.getFontNames("Times New Roman")) {
    System.out.println(fullName);
}
Times New Roman
Times New Roman Bold
Times New Roman Bold Italic
Times New Roman Italic
```

## Using Custom Fonts

You can load custom fonts from external sources: for example, from a file from the local file system or from a URL. The `loadFont()` static method in the `Font` class loads a custom font.

- `loadFont(InputStream in, double size)`
- `loadFont(String urlStr, double size)`

Upon successfully loading of the custom font, the `loadFont()` method registers font with JavaFX graphics engine, so a font can be created using the constructors and factory methods of the `Font` class. The method also creates a `Font` object of the specified `size` and returns it. Therefore, the `size` parameter exists for loading the font and creating its object in the same method call. If the method cannot load the font, it returns `null`.

The program in Listing 18-5 shows how to load a custom font from a local file system. The font file name is `4starfac.ttf`. The file was downloaded free from <http://www.fontfile.com>. The file is assumed to be in the CLASSPATH under `resources\font` directory. After the font is loaded successfully, it is set for the first `Text` node. A new `Font` object is created for its family name and set for the second `Text` node. If the font file does not exist or the font cannot be loaded, an appropriate error message is displayed in the window. Figure 18-6 shows the window when the font is loaded successfully.

### ***Listing 18-5.*** Loading and Using Custom Fonts Using the Font Class

```
// TextCustomFont.java
package com.jdojo.shape;

import java.net.URL;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.text.Font;
import javafx.scene.text.FontPosture;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class TextCustomFont extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Text t1 = new Text();
        t1.setLineSpacing(10);

        Text t2 = new Text("Another Text node");

        // Load the custom font
        String fontFile = "resources/font/4starfac.ttf";
        URL url
= this.getClass().getClassLoader().getResource(fontFile);
        if (url != null) {
            String urlString = url.toExternalForm();
            Font customFont = Font.loadFont(urlString, 16);
            if (customFont != null) {
                // Set the custom font for the first
Text node
                t1.setFont(customFont);

                // Set the text and line spacing
                t1.setText("Hello from the custom
font!!! \nFont Family: " +
                           customFont.getFamily());

                // Create an object of the custom font
and use it
                Font font2
= Font.font(customFont.getFamily(), FontWeight.BOLD,
                           FontPosture.ITALIC,
24);

                // Set the custom font for the second
Text node
                t2.setFont(font2);
            } else {

```

```

        t1.setText("Could not load the custom
font from " + urlStr);
    }
} else {
    t1.setText("Could not find the custom font
file " +
            fontFile + " in CLASSPATH. Used the
default font.");
}

HBox root = new HBox(t1, t2);
root.setSpacing(20);
root.setStyle("-fx-padding: 10;" +
              "-fx-border-style: solid inside;" +
              "-fx-border-width: 2;" +
              "-fx-border-insets: 5;" +
              "-fx-border-radius: 5;" +
              "-fx-border-color: blue;");

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("Loading and Using Custom Font");
stage.show();
}
}

```



**Figure 18-6.** Text nodes using custom fonts

## Setting Text Fill and Stroke

A Text node is a shape. Like a shape, it can have a fill and a stroke. By default, a Text node has `nullstroke` and `Color.BLACK` fill.

The Text class inherits properties and methods for setting its stroke and fill from the Shape class. I have discussed them at length in Chapter 17.

The Program in Listing 18-6 shows how to set stroke and fill for Text nodes. Figure 18-7 shows two Text nodes. The first one uses a red stroke and a white fill. The second one uses a black stroke and white fill. The stroke style for the second one uses a dashed line.

### **Listing 18-6.** Using Stroke and Fill for Text Nodes

```
// TextFillAndStroke.java
package com.jdojo.shape;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class TextFillAndStroke extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Text t1 = new Text("Stroke and fill!");
        t1.setStroke(Color.RED);
        t1.setFill(Color.WHITE);
        t1.setFont(new Font(36));

        Text t2 = new Text("Dashed Stroke!");
        t2.setStroke(Color.BLACK);
        t2.setFill(Color.WHITE);
        t2.setFont(new Font(36));
        t2.getStrokeDashArray().addAll(5.0, 5.0);

        HBox root = new HBox(t1, t2);
        root.setSpacing(20);
        root.setStyle("-fx-padding: 10;" +
                     "-fx-border-style: solid inside;" +
                     "-fx-border-width: 2;" +
                     "-fx-border-insets: 5;" +
                     "-fx-border-radius: 5;" +
                     "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using Stroke and Fill for Text Nodes");
        stage.show();
    }
}
```

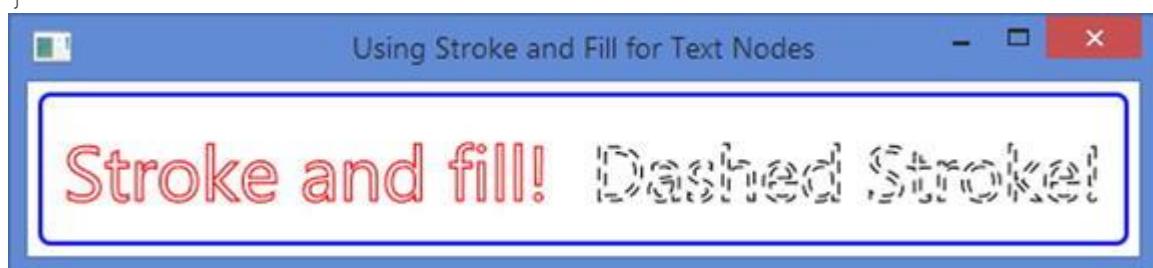


Figure 18-7. Text nodes using strokes and fills

## Applying Text Decorations

The `Text` class contains two boolean properties to apply text decorations to its text:

- `strikethrough`
- `underline`

By default, both properties are set to `false`. If the `strikethrough` is set to `true`, a line is drawn through each line of text. If the `underline` is set to `true`, a line is drawn below each line of text. The following snippet of code uses the decorations for `Text` nodes. The nodes are shown in Figure 18-8.

```
Text t1 = new Text("It uses the \nunderline decoration.");
t1.setUnderline(true);

Text t2 = new Text("It uses the \nstrikethrough decoration.");
t2.setStrikethrough(true);
It uses the underline decoration. It uses the strikethrough decoration.
```

**Figure 18-8.** Text nodes using the underline and strikethrough decorations

## Applying Font Smoothing

The `Text` class contains a `fontSmoothingType` property, which can be used to apply a gray or LCD font smoothing. Its value is one of the constants of the `FontSmoothingType` enum: `GRAY` and `LCD`. The default-smoothing type is `fontSmoothingType.GRAY`.

The `LCD` smoothing type is used as a hint. The following snippet of code creates two `Text` nodes: one uses `LCD` and one `GRAY` font-smoothing type. The `Text` nodes have been shown in Figure 18-9.

```
Text t1 = new Text("Hello world in LCD.");
t1.setFontSmoothingType(FontSmoothingType.LCD);

Text t2 = new Text("Hello world in GRAY.");
t2.setFontSmoothingType(FontSmoothingType.GRAY);
Hello world in LCD. Hello world in GRAY.
```

**Figure 18-9.** Text nodes using LCD and GRAY font-smoothing types

## Styling a Text Node with CSS

A `Text` node does not have a default CSS style-class name. In addition to all CSS properties of the `Shape`, a `Text` node supports the following CSS properties:

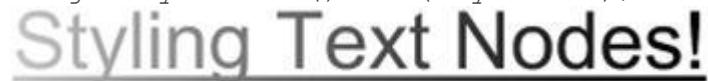
- -fx-font
- -fx-font-smoothing-type
- -fx-text-origin
- -fx-text-alignment
- -fx-strikethrough
- -fx-underline

I have discussed all properties in the previous sections. The `-fx-font` property is inherited from the parent. If the parent does not set the `-fx-font` property, the default system font is used. The valid values for the `-fx-font-smoothing-type` property are `lcd` and `gray`. The valid values for the `-fx-text-origin` property are `baseline`, `top`, and `bottom`. Let us create a style named `my-text` as follows. It sets a font and a linear gradient fill. The fill starts as a light gray color and ends as black.

```
.my-text {
    -fx-font: 36 Arial;
    -fx-fill: linear-gradient(from 0% 0% to 100% 0%, lightgray
0%, black 100%);
    -fx-font-smoothing-type: lcd;
    -fx-underline: true;
}
```

The following snippet of code creates a `Text` node and sets its style-class name to `my-text`. Figure 18-10 shows the `Text` node with its style applied to it.

```
Text t1 = new Text("Styling Text Nodes!");
t1.getStyleClass().add("my-text");
```



**Figure 18-10.** A `Text` node using CSS styles

## Summary

A `Text` node is an instance of the `Text` class that is used to render text. The `Text` class contains several properties to customize the appearance of text. The `Text` class and all its related classes are in the `javafx.scene.text` package. The `Text` class inherits from the `Shape` class. That is, a `Text` is a `Shape`, which allows you to use all properties and methods of the `Shape` class on a `Text` node. A `Text` node is capable of displaying multiple lines of text.

A `Text` node contains text and properties to render the text. You can create a `Text` node using one of the three constructors of the `Text` class. You can specify the text or text and position of the text while creating the node. The no-args constructor creates a `Text` node with an empty text and is located at (0, 0).

The no-args constructor creates a `Text` node with an empty string as its text. Other constructors let you specify the text and position the node. The `width` and `height` of a text node are automatically determined by its font. By default, a `Text` node uses a system default font to render its text.

Apart from the local and parent coordinate system, a `Text` node has an additional coordinate system. It is the coordinate system used for drawing the text. The `x`, `y`, and `textOrigin` properties of the `Text` class define the text coordinate system: The `x` and `y` properties define the `x` and `y` coordinates of the text origin.

The `textOrigin` property is of type `VPos`. Its value could be `VPos.BASELINE`, `VPos.TOP`, `VPos.CENTER`, and `VPos.BOTTOM`. The default is `VPos.BASELINE`. It defines where the `x`-axis of the text coordinate system lies within the text height.

The `font` property of the `Text` class defines the font for the text. The default font used is from “System” font family with the “Regular” style. The size of the default font is dependent on the platform and the desktop settings of the user. An instance of the `Font` class represents a font. The `Font` class contains several static methods that let you access the installed fonts on your computer and load custom fonts from font files.

A `Text` node is a shape. Like a shape, it can have a fill and a stroke. By default, a `Text` node has `nullstroke` and `Color.BLACK` fill.

The `strikethrough` and `underline` properties of the `Text` class lets you text decorations to the text. By default, both properties are set to `false`.

The `Text` class contains a `fontSmoothingType` property, which can be used to apply a gray or LCD font smoothing. Its value is one of the constants of the `FontSmoothingType` enum: `GRAY` and `LCD`. The default-smoothing type is `fontSmoothingType.GRAY`. The `LCD`-smoothing type is used as a hint.

You can style `Text` nodes using CSS. Setting font, text alignment, font smoothing, and decorations are supported through CSS.

The next chapter will discuss how to draw 3D shapes in JavaFX.

## CHAPTER 19



### Understanding 3D Shapes

In this chapter, you will learn:

- About 3D shapes and the classes representing 3D shapes in JavaFX
- How to check whether your machine supports 3D
- About the 3D coordinate system used in JavaFX
- About the rendering order of nodes
- How to draw predefined 3D shapes
- About the different types of cameras and how to use them to render scenes
- How to use light sources to view 3D objects in scenes
- How to create and use subscenes
- How to draw user-defined 3D shapes in JavaFX

### What Are 3D Shapes?

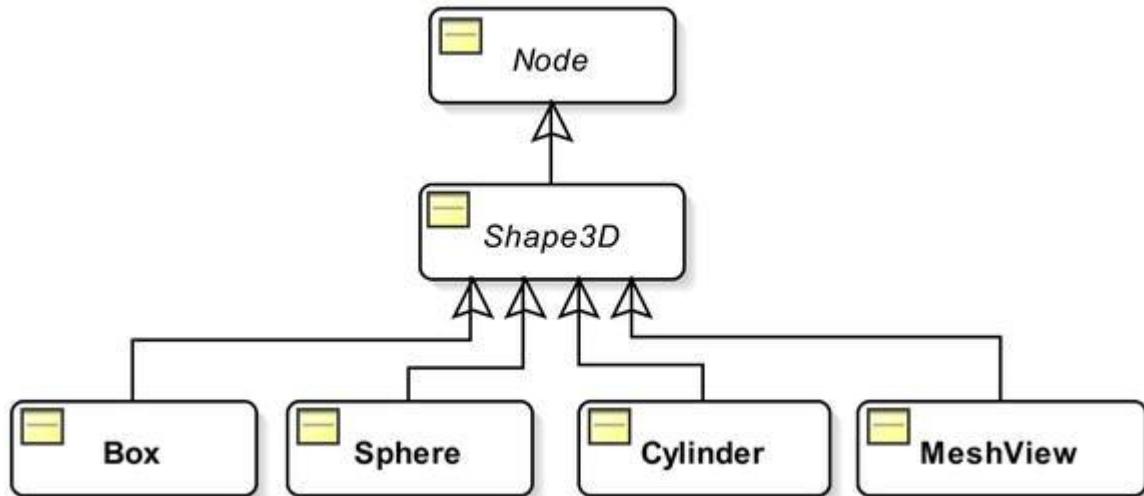
Any shape, drawn in a three-dimensional space, having three dimensions (length, width, and depth) is known as a 3D shape. Cubes, spheres, and pyramids are examples.

Although it was possible to have 2D nodes with 3D effects before, JavaFX 8 offers real 3D shapes as nodes. Before Java FX 8, the 3D effects were achieved using transformations in 3D space. JavaFX 8 offers two types of 3D shapes.

- Predefined shapes
- User-defined shapes

Box, sphere, and cylinder are three predefined 3D shapes that you can readily use in your JavaFX applications. You can also create any type of 3D shapes using a triangle mesh.

Figure 19-1 shows a class diagram of classes representing JavaFX 3D shapes. The 3D-shape classes are in the `javafx.scene.shape` package. The `Box`, `Sphere`, and `Cylinder` classes represent the three predefined shapes. The `MeshView` class represents a user-defined 3D shape in a scene.



**Figure 19-1.** A class diagram for classes representing 3D shapes

The 3D visualization in JavaFX is accomplished using lights and cameras. Lights and cameras are also nodes, which are added to the scene. You add 3D nodes to a scene, light it with lights, and view it using a camera. The positions of lights and cameras in the space determine the lighted and viewable areas of the scene. Figure 19-2 shows a 3D box, which is created using an instance of the `Box` class.



**Figure 19-2.** An example of a 3D box shape

## Checking Support for 3D

JavaFX 3D support is a conditional feature. If it is not supported on your platform, you get a warning message on the console when you run a program that attempts to use 3D features. Run the program in Listing 19-1 to check if your machine supports JavaFX 3D. The program will print a message stating whether the 3D support is available.

### **Listing 19-1.** Checking JavaFX 3D Support on Your Machine

```

// Check3DSupport.java
package com.jdojo.shape3d;

import javafx.application.ConditionalFeature;

```

```

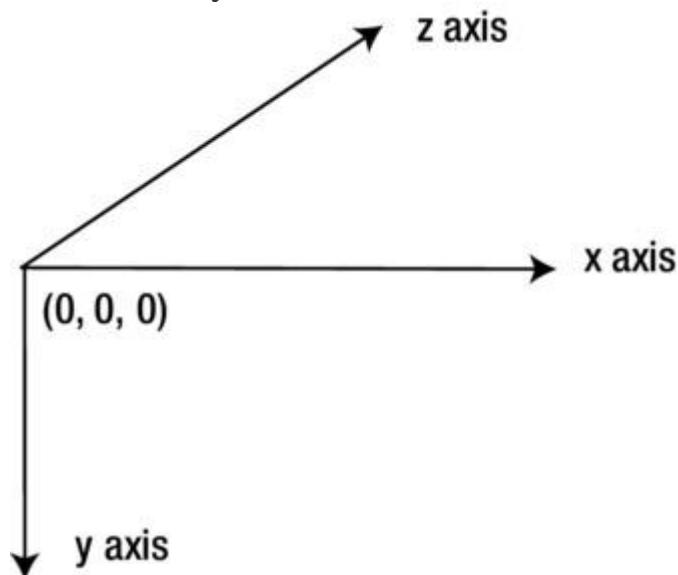
import javafx.application.Platform;

public class Check3DSupport {
    public static void main(String[] args) {
        boolean supported
= Platform.isSupported(ConditionalFeature.SCENE3D);
        if (supported) {
            System.out.println("3D is supported on your machine.");
        } else {
            System.out.println("3D is not supported on your
machine.");
        }
    }
}

```

## The 3D Coordinate System

A point in the 3D space is represented by (x, y, z) coordinates. A 3D object has three dimensions: x, y, and z. Figure 19-3 shows the 3D coordinate system used in JavaFX.



**Figure 19-3.** The 3D coordinate system used in JavaFX

The positive direction of the x-axis points to the right from the origin; the positive direction of the y-axis points down; the positive direction of the z-axis points into the screen (away from the viewer). The negative directions on the axes, which are not shown, extend in the opposite directions at the origin.

## Rendering Order of Nodes

Suppose you are looking at two overlapping objects at a distance. The object closer to you always overlaps the object farther from you, irrespective of the sequence in which they appeared in the view. When

dealing with 3D objects in JavaFX, you would like them to appear the same way.

In JavaFX, by default, nodes are rendered in the order they are added to the scene graph. Consider the following snippet of code:

```
Rectangle r1 = new Rectangle(0, 0, 100, 100);
Rectangle r2 = new Rectangle(50, 50, 100, 100);
Group group = new Group(r1, r2);
```

Two rectangles are added to a group. The rectangle `r1` is rendered first followed by rectangle `r2`. The overlapping area will show only the area of `r2`, not `r1`. If the group was created as `new Group(r2, r1)`, the rectangle `r2` will be rendered first followed with rectangle `r1`. The overlapping area will show the area of `r1`, not `r2`. Let us add the z coordinates for the two rectangles as follows:

```
Rectangle r1 = new Rectangle(0, 0, 100, 100);
r1.setTranslateZ(10);

Rectangle r2 = new Rectangle(50, 50, 100, 100);
r2.setTranslateZ(50);

Group group = new Group(r1, r2);
```

The foregoing snippet of code will produce the same effect as before. The rectangle `r1` will be rendered first followed by the rectangle `r2`. The z values for the rectangles are ignored. In this case, you would like to render the rectangle `r1` last as it is closer to the viewer (`z=10` is closer than `z=50`).

The previous rendering behavior is not desirable in a 3D space. You expect the 3D objects to appear the same way as they would appear in a real world. You need to do two things to achieve this.

- When creating a `Scene` object, specify that it needs to have a depth buffer.
- Specify in the nodes that their z coordinate values should be used during rendering. That is, they need to be rendered according to their depth (the distance from the viewer).

When you create a `Scene` object, you need to specify the `depthBuffer` flag, which is set to false by default.

```
// Create a Scene object with depthBuffer set to true
double width = 300;
double height = 200;
boolean depthBuffer = true;
Scene scene = new Scene(root, width, height, depthBuffer);
```

The `depthBuffer` flag for a scene cannot be changed after the scene is created. You can check whether a scene has a `depthBuffer` using the `isDepthBuffer()` method of the `Scene` object.

The `Node` class contains a `depthTest` property, which is available for all nodes in JavaFX. Its value is one of the constants of the `javafx.scene.DepthTest` enum:

- `ENABLE`
- `DISABLE`
- `INHERIT`

The `ENABLE` value for the `depthTest` indicates that the z coordinate values should be taken into account when the node is rendered. When the depth testing is enabled for a node, its z coordinate is compared with all other nodes with depth testing enabled, before rendering.

The `DISABLE` value indicates that the nodes are rendered in the order they are added to the scene graph.

The `INHERIT` value indicates that the `depthTest` property for a node is inherited from its parent. If a node has `null` parent, it is the same as `ENABLE`.

The program in Listing 19-2 demonstrates the concepts of using the depth buffer for a scene and the depth test for nodes. It adds two rectangles to a group. The rectangles are filled with red and green colors. The z coordinates for the red and green rectangles are 400px and 300px, respectively. The green rectangle is added to the group first. However, it is rendered first as it is closer to the viewer. You have added a camera to the scene, which is needed to view objects having depth (the z coordinate). The `CheckBox` is used to enable and disable the depth test for the rectangles. When the depth test is disabled, the rectangles are rendered in the order they are added to the group: the green rectangle followed with the red rectangle. Figure 19-4 shows rectangles in both states.

### *****Listing 19-2.***** Enabling/Disabling DepthTest Property for Nodes

```
// DepthTestCheck.java
package com.jdojo.shape3d;

import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.PerspectiveCamera;
import javafx.scene.Scene;
import javafx.scene.control.CheckBox;
import javafx.scene.layout.BorderPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.DepthTest;
import javafx.stage.Stage;
```

```

public class DepthTestCheck extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // Create two rectangles and add them to a Group
        Rectangle red = new Rectangle(100, 100);
        red.setFill(Color.RED);
        red.setTranslateX(100);
        red.setTranslateY(100);
        red.setTranslateZ(400);

        Rectangle green = new Rectangle(100, 100);
        green.setFill(Color.GREEN);
        green.setTranslateX(150);
        green.setTranslateY(150);
        green.setTranslateZ(300);

        Group center = new Group(green, red);

        CheckBox depthTestCbx = new CheckBox("DepthTest for
Rectangles");
        depthTestCbx.setSelected(true);
        depthTestCbx.selectedProperty().addListener(
            (prop, oldValue, newValue) -> {
                if (newValue) {
                    red.setDepthTest(DepthTest.ENABLE);
                    green.setDepthTest(DepthTest.ENABLE);
                }
                else {
                    red.setDepthTest(DepthTest.DISABLE);
                    green.setDepthTest(DepthTest.DISABLE);
                }
            });
        // Create a BorderPane as the root node for the
        scene. Need to
        // set the background transparent, so the camera can
        view the
        // rectangles behind the surface of the BorderPane
        BorderPane root = new BorderPane();
        root.setStyle("-fx-background-color: transparent;");
        root.setTop(depthTestCbx);
        root.setCenter(center);

        // Create a scene with depthBuffer enabled
        Scene scene = new Scene(root, 200, 200, true);

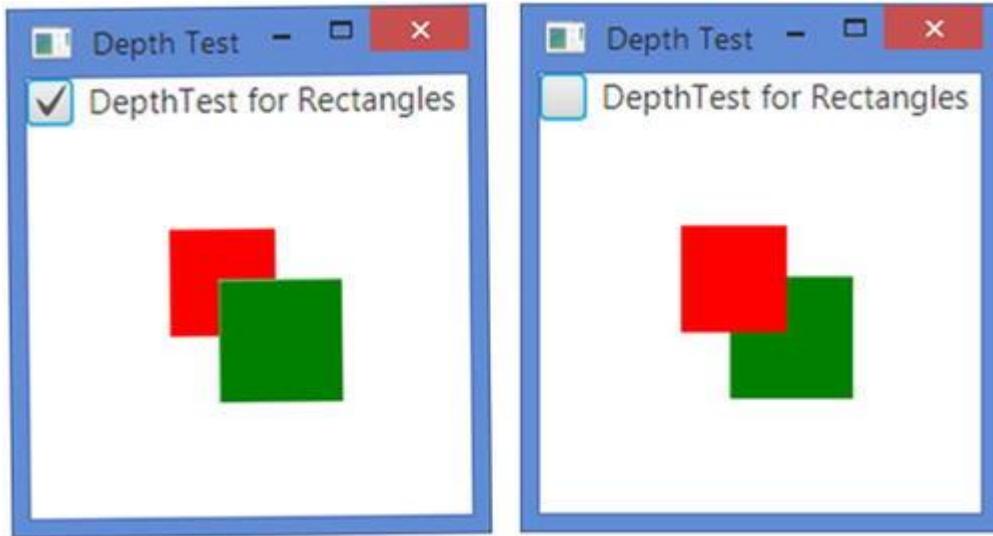
        // Need to set a camera to look into the 3D space of
        the scene
        scene.setCamera(new PerspectiveCamera());
        stage.setScene(scene);
    }
}

```

```

        stage.setTitle("Depth Test");
        stage.show();
    }
}

```



**Figure 19-4.** Effects of `depthTest` property on rendering nodes

## Using Predefined 3D Shapes

JavaFX 8 provides the following three built-in 3D geometric shapes:

- Box
- Sphere
- Cylinder

The shapes are represented by instances of the `Box`, `Sphere`, and `Cylinder` classes. The classes inherit from the `Shape3D` class, which contains three properties that are common to all types of 3D shapes.

- Material
- Draw mode
- Cull face

I will discuss these properties in detail in subsequent sections. If you do not specify these properties for a shape, reasonable defaults are provided.

The properties specific to a shape type are defined in the specific class defining the shape. For example, properties for a box are defined in the `Box` class. All shapes are nodes. Therefore, you can apply transformations to them. You can position them at any point in the 3D

space using the `translateX`, `translateY`, and `translateZ` transformations

**Tip** The center of a 3D shape is located at the origin of the local coordinate system of the shape.

A `Box` is defined by the following three properties:

- `width`
- `height`
- `depth`

The `Box` class contains two constructors:

- `Box()`
- `Box(double width, double height, double depth)`

The no-args constructor creates a `Box` with `width`, `height`, and `depth` of `2.0` each. The other constructor lets you specify the dimensions of the `Box`. The center of the `Box` is located at the origin of its local coordinate system.

```
// Create a Box with width=10, height=20, and depth=50
Box box = new Box(10, 20, 50);
```

A `Sphere` is defined by only one property named `radius`.

The `Sphere` class contains three constructors:

- `Sphere()`
- `Sphere(double radius)`
- `Sphere(double radius, int divisions)`

The no-args constructor creates a sphere of radius `1.0`.

The second constructor lets you specify the `radius` of the sphere.

The third constructor lets you specify the `radius` and `divisions`. A 3D sphere is made up of many divisions, which are constructed from connected triangles. The value of the number of divisions defines the resolution of the sphere. The higher the number of divisions, the smoother the sphere looks. By default, a value of `64` is used for the `divisions`. The value of `divisions` cannot be less than `1`.

```
// Create a Sphere with radius =50
Sphere sphere = new Sphere(50);
```

A `Cylinder` is defined by two properties:

- `radius`
- `height`

The radius of the cylinder is measured on the XZ plane. The axis of the cylinder is measured along the y-axis. The height of the cylinder is measured along its axis. The `Cylinder` class contains three constructors:

- `Cylinder()`
- `Cylinder(double radius, double height)`
- `Cylinder(double radius, double height, int divisions)`

The no-args constructor creates a `Cylinder` with a `1.0` radius and a `2.0` height.

The second constructor lets you specify the `radius` and `height` properties.

The third constructor lets you specify the number of divisions, which defines the resolution of the cylinder. The higher the number of divisions, the smoother the cylinder looks. Its default value is `15` along the x-axis and z-axis each. Its value cannot be less than `3`. If a value less than `3` is specified, a value of `3` is used. Note that the number of divisions does not apply along the y-axis. Suppose the number of divisions is `10`. It means that the vertical surface of the cylinder is created using `10` triangles. The height of the triangle will extend the entire height of the cylinder. The base of the cylinder will be created using `10` triangles.

```
// Create a cylinder with radius=40 and height=120
Cylinder cylinder = new Cylinder(40, 120);
```

The program in Listing 19-3 shows how to create 3D shapes. Figure 19-5 shows the shapes.

### ***Listing 19-3.*** Creating 3D Primitive Shapes: Box, Sphere, and Cylinder

```
// PreDefinedShapes.java
package com.jdojo.shape3d;

import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.PerspectiveCamera;
import javafx.scene.PointLight;
import javafx.scene.Scene;
import javafx.scene.shape.Box;
import javafx.scene.shape.Cylinder;
import javafx.scene.shape.Sphere;
import javafx.stage.Stage;

public class PreDefinedShapes extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }
}
```

```
@Override
public void start(Stage stage) {
    // Create a Box
    Box box = new Box(100, 100, 100);
    box.setTranslateX(150);
    box.setTranslateY(0);
    box.setTranslateZ(400);

    // Create a Sphere
    Sphere sphere = new Sphere(50);
    sphere.setTranslateX(300);
    sphere.setTranslateY(-5);
    sphere.setTranslateZ(400);

    // Create a cylinder
    Cylinder cylinder = new Cylinder(40, 120);
    cylinder.setTranslateX(500);
    cylinder.setTranslateY(-25);
    cylinder.setTranslateZ(600);

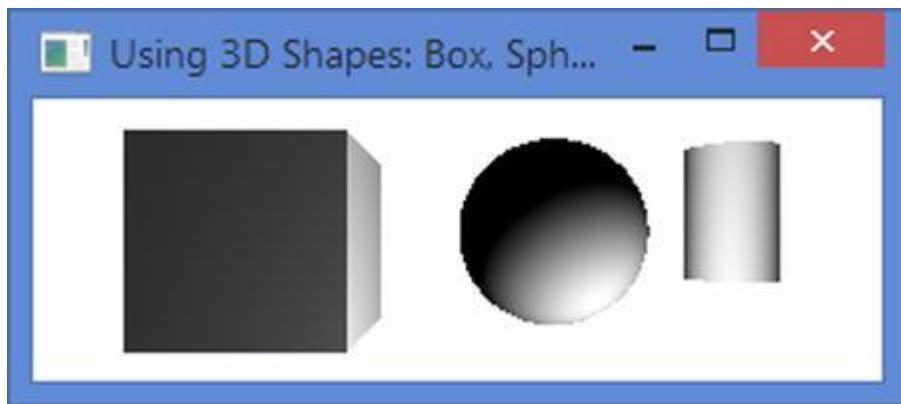
    // Create a light
    PointLight light = new PointLight();
    light.setTranslateX(350);
    light.setTranslateY(100);
    light.setTranslateZ(300);

    // Add shapes and a light to the group
    Group root = new Group(box, sphere, cylinder,
    light);

    // Create a Scene with depth buffer enabled
    Scene scene = new Scene(root, 300, 100, true);

    // Set a camera to view the 3D shapes
    PerspectiveCamera camera = new
    PerspectiveCamera(false);
    camera.setTranslateX(100);
    camera.setTranslateY(-50);
    camera.setTranslateZ(300);
    scene.setCamera(camera);

    stage.setScene(scene);
    stage.setTitle("Using 3D Shapes: Box, Sphere and
Cylinder");
    stage.show();
}
}
```



**Figure 19-5.** Primitive 3D shapes: a box, a sphere, and a cylinder

The program creates the three shapes and positions them in the space. It creates a light, which is an instance of the `PointLight`, and positions it in the space. Note that a light is also a `Node`. The light is used to light the 3D shapes. All shapes and the light are added to a group, which is added to the scene.

To view the shapes, you need to add a camera to the scene. The program adds a `PerspectiveCamera` to the scene. Note that you need to position the camera as its position and orientation in the space determine what you see. The origin of the local coordinate system of the camera is located at the center of the scene. Try resizing the window after you run the program. You will notice that the view of the shapes changes as you resize the window. It happens because the center of the scene is changing when you resize the window, which in turn repositions the camera, resulting in the change in the view.

### Specifying the Shape Material

A material is used for rendering the surface of shapes. You can specify the material for the surface of 3D objects using the `material` property, which is defined in the `Shape3D` class. The `material` property is an instance of the abstract class `Material`. JavaFX provides the `PhongMaterial` class as the only concrete implementation of `Material`. Both classes are in the `javafx.scene.paint` package. An instance of the `PhongMaterial` class represents Phong shaded material. Phong shaded material is based on Phong shading and the Phong reflection model (also known as Phong illumination and Phong lighting), which were developed at the University of Utah by Bui Tuong Phong as part of his Ph.D. dissertation in 1973. A complete discussion of the Phong model is beyond the scope of this book. The model provides an empirical formula to compute the color of a pixel on the geometric

surface in terms of the following properties defined in the PhongMaterial class:

- diffuseColor
- diffuseMap
- specularColor
- specularMap
- selfIlluminationMap
- specularPower
- bumpMap

The PhongMaterial class contains three constructors:

- PhongMaterial()
- PhongMaterial(Color diffuseColor)
- PhongMaterial(Color diffuseColor, Image diffuseMap, Image specularMap, Image bumpMap, Image selfIlluminationMap)

The no-args constructor creates a PhongMaterial with the diffuse color as Color.WHITE. The other two constructors are used to create a PhongMaterial with the specified properties.

When you do not provide a material for a 3D shape, a default material with a diffuse color of Color.LIGHTGRAY is used for rendering the shape. All shapes in our previous example in Listing 19-3 used the default material.

The following snippet of code creates a Box, creates a PhongMaterial with tan diffuse color, and sets the material to the box:

```
Box box = new Box(100, 100, 100);
PhongMaterial material = new PhongMaterial();
material.setDiffuseColor(Color.TAN);
box.setMaterial(material);
```

You can use an Image as the diffuse map to have texture for the material, as shown in the following code:

```
Box boxWithTexture = new Box(100, 100, 100);
PhongMaterial textureMaterial = new PhongMaterial();
Image randomness = new Image("resources/picture/randomness.jpg");
textureMaterial.setDiffuseMap(randomness);
boxWithTexture.setMaterial(textureMaterial);
```

The program in Listing 19-4 shows how to create and set material for shapes. It creates two boxes. It sets the diffuse color for one box and the diffuse map for other. The image used for the diffuse map provides the

texture for the surface of the second box. The two boxes look as shown in Figure 19-6.

#### ***Listing 19-4.*** Using the Diffuse Color and Diffuse Map to Create PhongMaterial

```
// MaterialTest.java
package com.jdojo.shape3d;

import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.PerspectiveCamera;
import javafx.scene.PointLight;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.paint.Color;
import javafx.scene.paint.PhongMaterial;
import javafx.scene.shape.Box;
import javafx.stage.Stage;

public class MaterialTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // Create a Box
        Box box = new Box(100, 100, 100);

        // Set the material for the box
        PhongMaterial material = new PhongMaterial();
        material.setDiffuseColor(Color.TAN);
        box.setMaterial(material);

        // Place the box in the space
        box.setTranslateX(250);
        box.setTranslateY(0);
        box.setTranslateZ(400);

        // Create a Box with texture
        Box boxWithTexture = new Box(100, 100, 100);
        PhongMaterial textureMaterial = new PhongMaterial();
        Image randomness = new
Image("resources/picture/randomness.jpg");
        textureMaterial.setDiffuseMap(randomness);
        boxWithTexture.setMaterial(textureMaterial);

        // Place the box in the space
        boxWithTexture.setTranslateX(450);
        boxWithTexture.setTranslateY(-5);
        boxWithTexture.setTranslateZ(400);

        PointLight light = new PointLight();
```

```

        light.setTranslateX(250);
        light.setTranslateY(100);
        light.setTranslateZ(300);

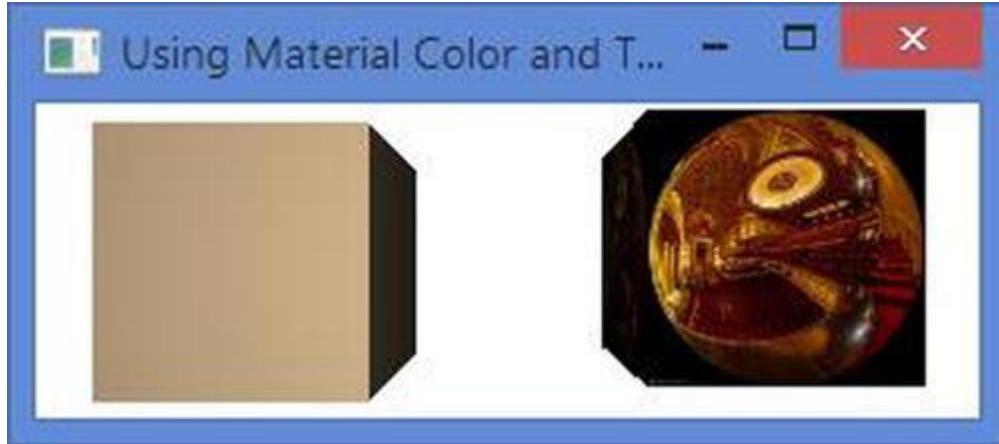
        Group root = new Group(box, boxWithTexture);

        // Create a Scene with depth buffer enabled
        Scene scene = new Scene(root, 300, 100, true);

        // Set a camera to view the 3D shapes
        PerspectiveCamera camera = new
PerspectiveCamera(false);
        camera.setTranslateX(200);
        camera.setTranslateY(-50);
        camera.setTranslateZ(325);
        scene.setCamera(camera);

        stage.setScene(scene);
        stage.setTitle("Using Material Color and Texture for
3D Surface");
        stage.show();
    }
}

```



**Figure 19-6.** Two boxes: one with a tan diffuse color and one with texture using a diffuse map

### Specifying the Draw Mode of Shapes

A 3D shape surface consists of many connected polygons made up of triangles. For example, a Box is made up of 12 triangles—each side of the Box using two triangles. The `drawMode` property in the `Shape3D` class specifies how the surface of 3D shapes is rendered. Its value is one of the constants of the `DrawMode` enum.

- `DrawMode.FILL`
- `DrawMode.LINE`

The `DrawMode.FILL` is the default and it fills the interior of the triangles. The `DrawMode.LINE` draws only the outline of the triangles. That is, it draws only lines connecting the vertices of the consecutive triangles.

```
// Create a Box with outline only
Box box = new Box(100, 100, 100);
box.setDrawMode(DrawMode.LINE);
```

The program in Listing 19-5 shows how to draw only the outline of 3D shapes. Figure 19-7 shows the shapes. The program is similar to the one shown in Listing 19-3. The program sets the `drawMode` property of all shapes to `DrawMode.LINE`. The program specifies the divisions of creating the `Sphere` and `Cylinder`. Change the value for divisions to a lesser value. You will notice that the number of triangles used to create the shapes decreases, making the shape less smooth.

### ***Listing 19-5.*** Drawing Only Lines for 3D Shapes

```
// DrawModeTest.java
package com.jdojo.shape3d;

import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.PerspectiveCamera;
import javafx.scene.PointLight;
import javafx.scene.Scene;
import javafx.scene.shape.Box;
import javafx.scene.shape.Cylinder;
import javafx.scene.shape.DrawMode;
import javafx.scene.shape.Sphere;
import javafx.stage.Stage;

public class DrawModeTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // Create a Box
        Box box = new Box(100, 100, 100);
        box.setDrawMode(DrawMode.LINE);
        box.setTranslateX(150);
        box.setTranslateY(0);
        box.setTranslateZ(400);

        // Create a Sphere: radius = 50, divisions=20
        Sphere sphere = new Sphere(50, 20);
        sphere.setDrawMode(DrawMode.LINE);
        sphere.setTranslateX(300);
        sphere.setTranslateY(-5);
    }
}
```

```

        sphere.setTranslateZ(400);

        // Create a cylinder: radius=40, height=120,
        divisions=5
        Cylinder cylinder = new Cylinder(40, 120, 5);
        cylinder.setDrawMode(DrawMode.LINE);
        cylinder.setTranslateX(500);
        cylinder.setTranslateY(-25);
        cylinder.setTranslateZ(600);

        PointLight light = new PointLight();
        light.setTranslateX(350);
        light.setTranslateY(100);
        light.setTranslateZ(300);

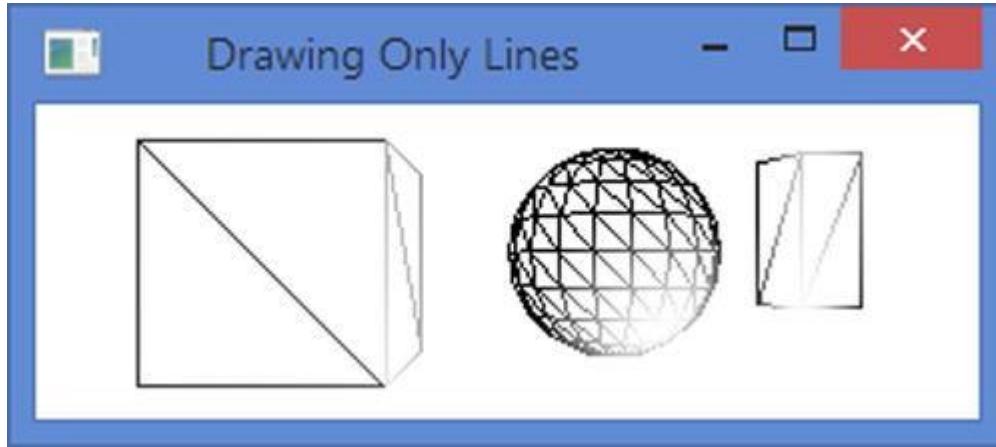
        Group root = new Group(box, sphere, cylinder,
        light);

        // Create a Scene with depth buffer enabled
        Scene scene = new Scene(root, 300, 100, true);

        // Set a camera to view the 3D shapes
        PerspectiveCamera camera = new
        PerspectiveCamera(false);
        camera.setTranslateX(100);
        camera.setTranslateY(-50);
        camera.setTranslateZ(300);
        scene.setCamera(camera);

        stage.setScene(scene);
        stage.setTitle("Drawing Only Lines");
        stage.show();
    }
}

```



**Figure 19-7.** Drawing the outline of 3D shapes

### Specifying the Face Culling for Shapes

A 3D object is never visible entirely. For example, you can never see an entire building at once. When you change the viewing angle, you see

different parts of the building. If you face the front of the building, you see only the front part of the building. Standing in front, if you move to the right, you see the front and right sides of the building.

The surface of 3D objects is made of connected triangles. Each triangle has two faces: the exterior face and the interior face. You see the exterior face of the triangles when you look at the 3D objects. Not all triangles are visible all the time. Whether a triangle is visible depends on the position of the camera. There is a simple rule to determine the visibility of triangles making up the surface of a 3D object. Draw a line coming out from the plane of the triangle and the line is perpendicular to the plane of a triangle. Draw another line from the point where the first line intersects the plane of the triangle to the viewer. If the angle between two lines is greater than 90 degrees, the face of the triangle is not visible to the view. Otherwise, the face of the triangle is visible to the viewer. Note that not both faces of a triangle are visible at the same time.

Face culling is a technique of rendering 3D geometry based on the principle that the nonvisible parts of an object should not be rendered. For example, if you are facing a building from the front, there is no need to render the sides, top, and bottom of the building, as you cannot see them.

**Tip** Face culling is used in 3D rendering to enhance performance.

The `Shape3D` class contains a `cullFace` property that specifies the type of culling applied in rendering the shape. Its value is one of the constants of the `CullFace` enum:

- BACK
- FRONT
- NONE

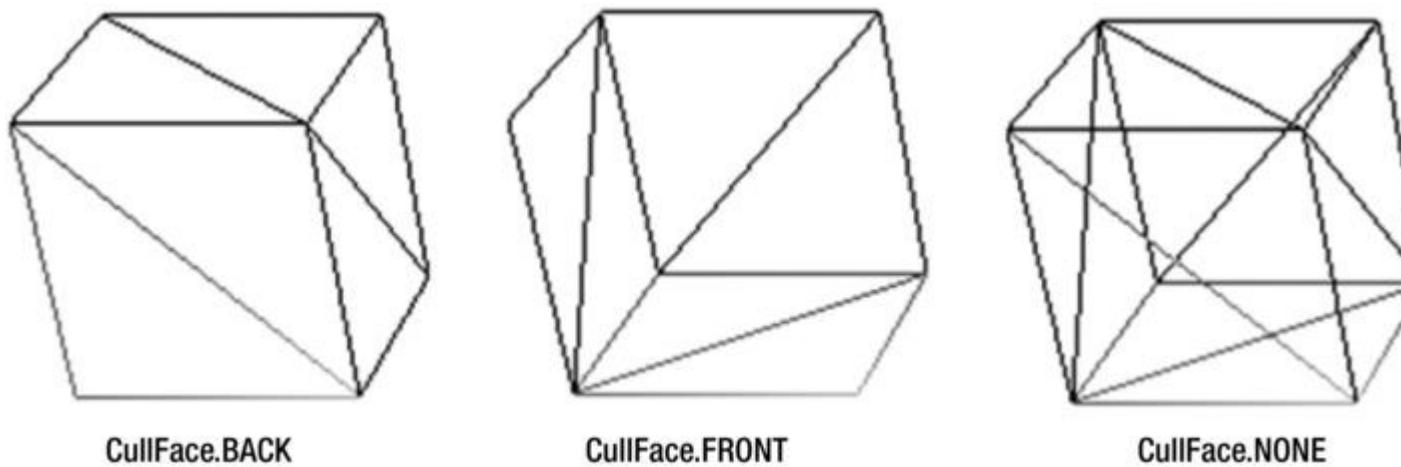
The `CullFace.BACK` specifies that all triangles that cannot be seen through the camera in its current position should be culled (i.e., not rendered). That is, all triangles whose exterior faces are not facing the camera should be culled. If you are facing the front of a building, this setting will render only the front part of the building. This is the default.

The `CullFace.FRONT` specifies that the all triangles whose exterior faces are facing the camera should be culled. If you are facing the front of a building, this setting will render all parts of the building, except the front part.

The `CullFace.NONE` specifies that no face culling should be applied. That is, all triangles making up the shape should be rendered.

```
// Create a Box with no face culling
Box box = new Box(100, 100, 100);
Box.setCullFace(CullFace.NONE);
```

It is easy to see the effect of face culling when you draw the shape using the `drawMode` as `DrawMode.LINE`. I will draw only nonculled triangles. Figure 19-8 shows the same Box using three different face cullings. The first Box (from left) uses the back-face culling, the second front-face culling, and the third one uses no culling. Notice that the first picture of the Box shows the front, right, and top faces whereas these faces are culled in the second Box. In the second picture, you see the back, left, and bottom faces. Note that when you use front-face culling, you see the interior faces of the triangles as the exterior faces are hidden from the view.



**Figure 19-8.** A box using different `cullFace` properties

## Using Cameras

Cameras are used to render the scene. Two types of cameras are available.

- Perspective camera
- Parallel camera

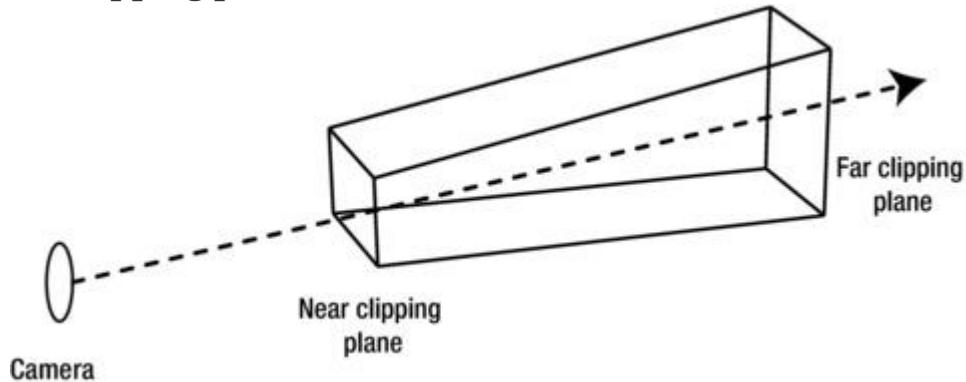
The names of the cameras suggest the projection type they use to render the scene. Cameras in JavaFX are nodes. They can be added to the scene graph and positioned like other nodes.

The abstract base class `Camera` represents a camera. Two concrete subclasses of the `Camera` class

exist: `PerspectiveCamera` and `ParallelCamera`. The three classes are in the `javafx.scene` package.

**Tip** Before Java 8, camera classes were inherited from the `Object` class and they were not nodes. In JavaFX 8, they inherit from the `Node` class.

A PerspectiveCamera defines the viewing volume for a perspective projection, which is a truncated right pyramid as shown in Figure 19-9. The camera projects the objects contained within the near and far clipping planes onto the projection plane. Therefore, any objects outside the clipping planes are not visible.



**Figure 19-9.** The viewing volume of a perspective camera defined by the near clip and far clip planes

The content that the camera will project onto the projection plane is defined by two properties in the Camera class.

- `nearClip`
- `farClip`

The `nearClip` is the distance between the camera and the near clipping plane. Objects closer to the camera than the `nearClip` are not rendered. The default value is 0.1.

The `farClip` is the distance between the camera and the far clipping plane. Objects farther from the camera than the `farClip` are not rendered. The default value is 100.

The PerspectiveCamera class contains two constructors.

- `PerspectiveCamera()`
- `PerspectiveCamera(boolean fixedEyeAtCameraZero)`

The no-args constructor creates a PerspectiveCamera with the `fixedEyeAtCameraZero` flag set to false, which makes it behave more or less like a parallel camera where the objects in the scene at Z=0 stay the same size when the scene is resized. The second constructor lets you specify this flag. If you want to view 3D objects with real 3D effects, you need to set this flag to true. Setting this flag to true will adjust the size of the projected images of the 3D objects as the scene is resized. Making the scene smaller will make the objects look smaller as well.

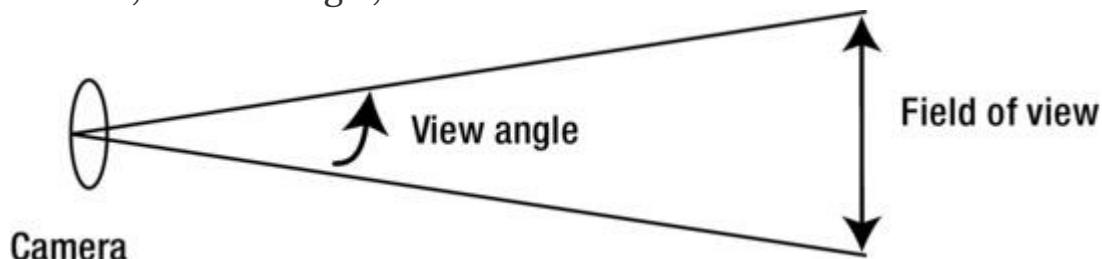
```
// Create a perspective camera for viewing 3D objects
PerspectiveCamera camera = new PerspectiveCamera(true);
```

The PerspectiveCamera class declares two additional properties.

- fieldOfView
- verticalFieldOfView

The fieldOfView is measured in degrees and it is the view angle of the camera. Its default value is 30 degrees.

The verticalFieldOfView property specifies whether the fieldOfView property applies to the vertical dimension of the projection plane. By default, its value is true. Figure 19-10 depicts the camera, its view angle, and field of view.



**Figure 19-10.** The view angle and field of view for a perspective camera

An instance of the ParallelCamera specifies the viewing volume for a parallel projection, which is a rectangular box.

The ParallelCamera class does not declare any additional properties. It contains a no-args constructor.

```
ParallelCamera camera = new ParallelCamera();
```

You can set a camera for a scene using the setCamera() method of the Scene class.

```
Scene scene = create a scene....  
PerspectiveCamera camera = new PerspectiveCamera(true);  
scene.setCamera(camera);
```

Because a camera is a node, you can add it to the scene graph.

```
PerspectiveCamera camera = new PerspectiveCamera(true);  
Group group = new Group(camera);
```

You can move and rotate the camera as you move and rotate nodes. To move it to a different position, use the translateX, translateY, and translateZ properties. To rotate, use the Rotate transformation.

The program in Listing 19-6 uses a PerspectiveCamera to view a Box. You have used two lights: one to light the front and the top faces and one to light the bottom face of the box. The camera is animated by rotating it indefinitely along the x-axis. As the camera rotates, it brings different parts of the box into the view. You can see the effect of the two

lights when the bottom of the box comes into the view. The bottom is shown in green whereas the top and front are in red.

### ***Listing 19-6.*** Using a PerspectiveCamera as a Node

```
// CameraTest.java
package com.jdojo.shape3d;

import javafx.animation.Animation;
import javafx.animation.RotateTransition;
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.PerspectiveCamera;
import javafx.scene.PointLight;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Box;
import javafx.scene.shape.CullFace;
import javafx.scene.transform.Rotate;
import javafx.stage.Stage;
import javafx.util.Duration;

public class CameraTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Box box = new Box(100, 100, 100);
        box.setCullFace(CullFace.NONE);
        box.setTranslateX(250);
        box.setTranslateY(100);
        box.setTranslateZ(400);

        PerspectiveCamera camera = new
PerspectiveCamera(false);
        camera.setTranslateX(100);
        camera.setTranslateY(-50);
        camera.setTranslateZ(300);

        // Add a Rotation animation to the camera
        RotateTransition rt = new
RotateTransition(Duration.seconds(2), camera);
        rt.setCycleCount(Animation.INDEFINITE);
        rt.setFromAngle(0);
        rt.setToAngle(90);
        rt.setAutoReverse(true);
        rt.setAxis(Rotate.X_AXIS);
        rt.play();

        PointLight redLight = new PointLight();
        redLight.setColor(Color.RED);
```

```

        redLight.setTranslateX(250);
        redLight.setTranslateY(-100);
        redLight.setTranslateZ(250);

        PointLight greenLight = new PointLight();
        greenLight.setColor(Color.GREEN);
        greenLight.setTranslateX(250);
        greenLight.setTranslateY(300);
        greenLight.setTranslateZ(300);

        Group root = new Group(box, redLight, greenLight);
        root.setRotationAxis(Rotate.X_AXIS);
        root.setRotate(30);

        Scene scene = new Scene(root, 500, 300, true);
        scene.setCamera(camera);
        stage.setScene(scene);
        stage.setTitle("Using cameras");
        stage.show();
    }
}

```

## Using Light Sources

Similar to the real world, you need a light source to view the 3D objects in a scene. An instance of the abstract base class `LightBase` represents a light source. Its two concrete subclasses, `AmbientLight` and `PointLight`, represent an ambient light and a point light. Light source classes are in the `javafx.scene` package. The `LightBase` class inherits from the `Node` class. Therefore, a light source is a node and it can be added to the scene graph as any other nodes.

A light source has three properties: light color, on/off switch, and a list of affected nodes. The `LightBase` class contains the following two properties:

- `color`
- `lightOn`

The `color` specifies the color of the light. The `lightOn` specifies whether the light is on. The `getScope()` method of the `LightBase` class returns an `ObservableList<Node>`, which is the hierarchical list of nodes affected by this light source. If the list is empty, the scope of the light source is universe, which means that it affects all nodes in the scene.

An instance of the `AmbientLight` class represents an ambient light source. An ambient light is a nondirectional light that seems to come

from all directions. Its intensity is constant on the surface of the affected shapes.

```
// Create a red ambient light
AmbientLight redLight = new AmbientLight(Color.RED);
```

An instance of the `PointLight` class represents a point light source. A point light source is a fixed point in space and radiates lights equally in all directions. The intensity of a point light decreases as the distance of the lighted point increases from the light source.

```
// Create a Add the point light to a group
PointLight redLight = new PointLight(Color.RED);
redLight.setTranslateX(250);
redLight.setTranslateY(-100);
redLight.setTranslateZ(290);
Group group = new Group(node1, node2, redLight);
```

## Creating Subscenes

A scene can use only one camera. Sometimes, you may want to view different parts of a scene using multiple cameras. JavaFX 8 introduces the concept as subscenes. A subscene is a container for a scene graph. It can have its own width, height, fill color, depth buffer, antialiasing flag, and camera. An instance of the `SubScene` class represents a subscene. The `SubScene` inherits from the `Node` class. Therefore, a subscene can be used wherever a node can be used. A subscene can be used to separate 2D and 3D nodes in an application. You can use a camera for the subscene to view 3D objects that will not affect the 2D nodes in the other part of the main scene. The following snippet of code creates a `SubScene` and sets a camera to it:

```
SubScene ss = new SubScene(root, 200, 200, true,
SceneAntialiasing.BALANCED);
PerspectiveCamera camera = new PerspectiveCamera(false);
ss.setCamera(camera);
```

**Tip** If a `SubScene` contains `Shape3D` nodes having a light node, a head light with a `PointLight` with `Color.WHITE` light source is provided. The head light is positioned at the camera position.

The program in Listing 19-7 shows how to use subscenes. The `getSubScene()` method creates a `SubScene` with a `Box`, a `PerspectiveCamera`, and a `PointLight`. An animation is set up to rotate the camera along the specified axis. The `start()` method creates two subscenes and adds them to an `HBox`. One subscene swings the camera along the y-axis and another along the x-axis. The `HBox` is added to the main scene.

### **Listing 19-7.** Using Subscenes

```
// SubSceneTest.java
package com.jdojo.shape3d;

import javafx.animation.Animation;
import javafx.animation.RotateTransition;
import javafx.application.Application;
import javafx.geometry.Point3D;
import javafx.scene.Group;
import javafx.scene.PerspectiveCamera;
import javafx.scene.PointLight;
import javafx.scene.Scene;
import javafx.scene.SceneAntialiasing;
import javafx.scene.SubScene;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Box;
import javafx.scene.shape.CullFace;
import javafx.scene.transform.Rotate;
import javafx.stage.Stage;
import javafx.util.Duration;

public class SubSceneTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        SubScene ySwing = getSubScene(Rotate.Y_AXIS);
        SubScene xSwing = getSubScene(Rotate.X_AXIS);
        HBox root = new HBox(20, ySwing, xSwing);
        Scene scene = new Scene(root, 500, 300, true);
        stage.setScene(scene);
        stage.setTitle("Using Sub-Scenes");
        stage.show();
    }

    private SubScene getSubScene(Point3D rotationAxis) {
        Box box = new Box(100, 100, 100);
        box.setCullFace(CullFace.NONE);
        box.setTranslateX(250);
        box.setTranslateY(100);
        box.setTranslateZ(400);

        PerspectiveCamera camera = new
PerspectiveCamera(false);
        camera.setTranslateX(100);
        camera.setTranslateY(-50);
        camera.setTranslateZ(300);

        // Add a Rotation animation to the camera
        RotateTransition rt = new
RotateTransition(Duration.seconds(2), camera);
        rt.setCycleCount(Animation.INDEFINITE);
    }
}
```

```

        rt.setFromAngle(-10);
        rt.setToAngle(10);
        rt.setAutoReverse(true);
        rt.setAxis(rotationAxis);
        rt.play();

        PointLight redLight = new PointLight(Color.RED);
        redLight.setTranslateX(250);
        redLight.setTranslateY(-100);
        redLight.setTranslateZ(290);

        // If you remove the redLight from the following
group,
        // a default head light will be provided by the
SubScene.
        Group root = new Group(box, redLight);
        root.setRotationAxis(Rotate.X_AXIS);
        root.setRotate(30);

        SubScene ss = new SubScene(root, 200, 200, true,
SceneAntialiasing.BALANCED);
        ss.setCamera(camera);
        return ss;
    }
}

```

## Creating User-Defined Shapes

JavaFX lets you define a 3D shape using a mesh of polygons. An instance of the abstract `Mesh` class represents the mesh data.

The `TriangleMesh` class is concrete subclass of the `Mesh` class. A `TriangleMesh` represents a 3D surface consisting of a mesh of triangles.

**Tip** In 3D modeling, a mesh of different types of polygons can be used to construct a 3D object. JavaFX supports only a mesh of triangles.

An instance of the `MeshView` class represents a 3D surface. The data for constructing a `MeshView` is specified as an instance of the `Mesh`.

Supplying the mesh data by hand is not an easy task. The problem is complicated by the way you need to specify the data. I will make it easier by demonstrating the mesh usage from a very simple user case to a more complex one.

A `TriangleMesh` needs to supply data for three aspects of a 3D object.

- Points
- Texture coordinates
- Faces

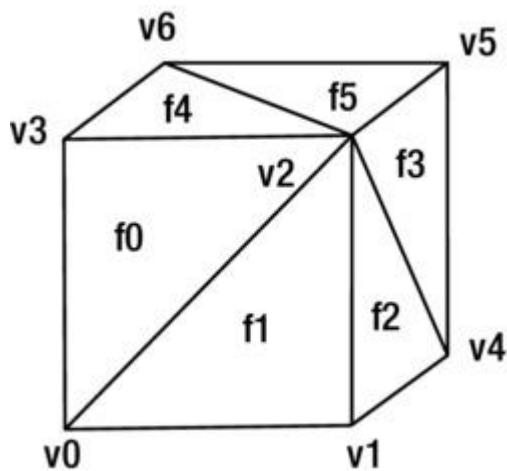
**Note** If you have not worked with 3D objects using a mesh of triangles before, the explanation may seem a little complex. You need to be patient and learn a step at a time to understand the process of creating a 3D object using a mesh of triangles.

Points are the vertices of the triangles in the mesh. You need to specify the (x, y, z) coordinates of vertices in an array. Suppose  $v_0, v_1, v_2, v_3, v_4$ , and so on are the points in 3D space that represent the vertices of the triangles in a mesh. Points in a `TriangleMesh` are specified as an array of floats.

The texture of a 3D surface is provided as an image that is a 2D object. Texture coordinates are points in a 2D plane, which are mapped to the vertices of triangles. You need to think of the triangles in a mesh unwrapped and placed onto a 2D plane. Overlay the image that supplies the surface texture for the 3D shape onto the same 2D plane. Map the vertices of the triangles to the 2D coordinates of the image to get a pair of (u, v) coordinates for each vertex in the mesh. The array of such (u, v) coordinates is the texture coordinate. Suppose  $t_0, t_1, t_2, t_3, t_4$ , and so on are the texture coordinates.

Faces are the planes created by joining the three edges of the triangles. Each triangle has two faces: a front face and a back face. A face is specified in terms of indices in the points and texture coordinates arrays. A face is specified as  $v_0, t_0, v_1, t_1, v_2, t_2$ , and so on, where  $v_1$  is the index of the vertex in the points array and  $t_1$  is the index of the vertex in the texture coordinates array.

Consider the box shown in Figure 19-11.



**Figure 19-11.** A box made of 12 triangles

A box consists of six sides. Each side is a rectangle. Each rectangle consists of two triangles. Each triangle has two faces: a front face and a back face. A box has eight vertices. You have named vertices as  $v_0, v_1, v_2, v_3, v_4, v_5, v_6$ .

and so on, and the faces as  $f_0, f_1, f_2$ , and so on in the figure. You do not see the numberings for the vertices and faces that are not visible in the current orientation of the box. Each vertex is defined by a triple  $(x, y, z)$ , which is the coordinate of the vertex in the 3D space. When you use the term *vertex  $v_1$* , you, technically, mean its coordinates  $(x_1, y_1, z_1)$  for the vertex.

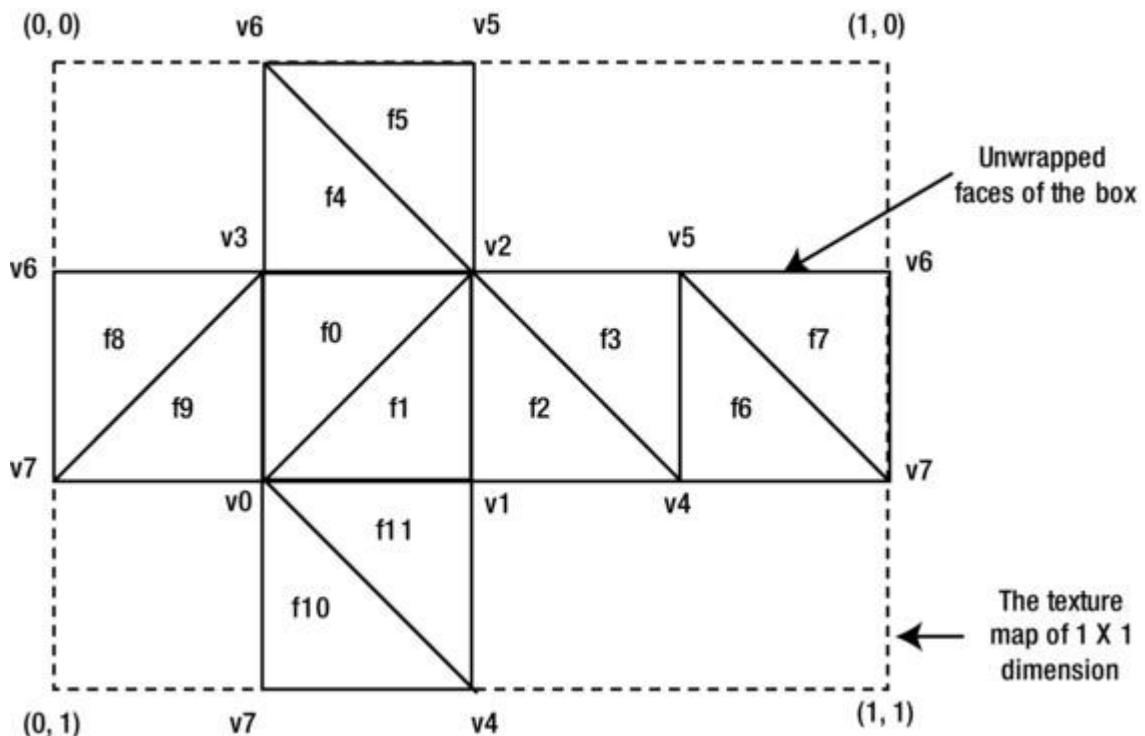
To create a mesh of triangles, you need to specify all vertices making up the 3D object. In the case of a box, you need to specify the eight vertices. In the `TriangleMesh` class, the vertices are known as `points` and they are specified as an observable array of `float`. The following pseudo-code creates the array of vertices. The first array is for understanding purpose only. The actual array specifies the coordinates of the vertices.

```
// For understanding purpose only
float[] points = {v0,
                  v1,
                  v2,
                  ...
                  v7};

// The actual array contain (x, y, z) coordinates of all vertices
float[] points = {x0, y0, z0, // v0
                  x1, y1, z1, // v1
                  x2, y2, z2, // v2
                  ...
                  x7, y7, z7 // v7
};
```

In the `points` array, the indices 0 to 2 contain coordinates of the first vertex, indices 3 to 5 contain the coordinates of the second vertex, and so on. How do you number the vertices? That is, which vertex is #1 and which one is #2, and so on? There is no rule to specify the order to vertices. It is all up to you how you number them. JavaFX cares about only one thing: you must include all vertices making up the shape in the `points` array. You are done with generating the `points` array. You will use it later.

Now, you need to create an array containing coordinates of 2D points. Creating this array is a little tricky. Beginners have hard time understanding this. Consider the figure shown in Figure 19-12.



**Figure 19-12.** Surface of a box mapped onto a 2D plane

Figure 19-11 and Figure 19-12 are two views of the surface of the same box. Figure 19-12 mapped the surface from the 3D space to a 2D plane. Think of the box as a 3D object made of 12 triangular pieces of paper. Figure 19-11 shows those 12 pieces of paper put together as a 3D box whereas Figure 19-12 shows the same pieces of paper put side by side on the floor (a 2D plane).

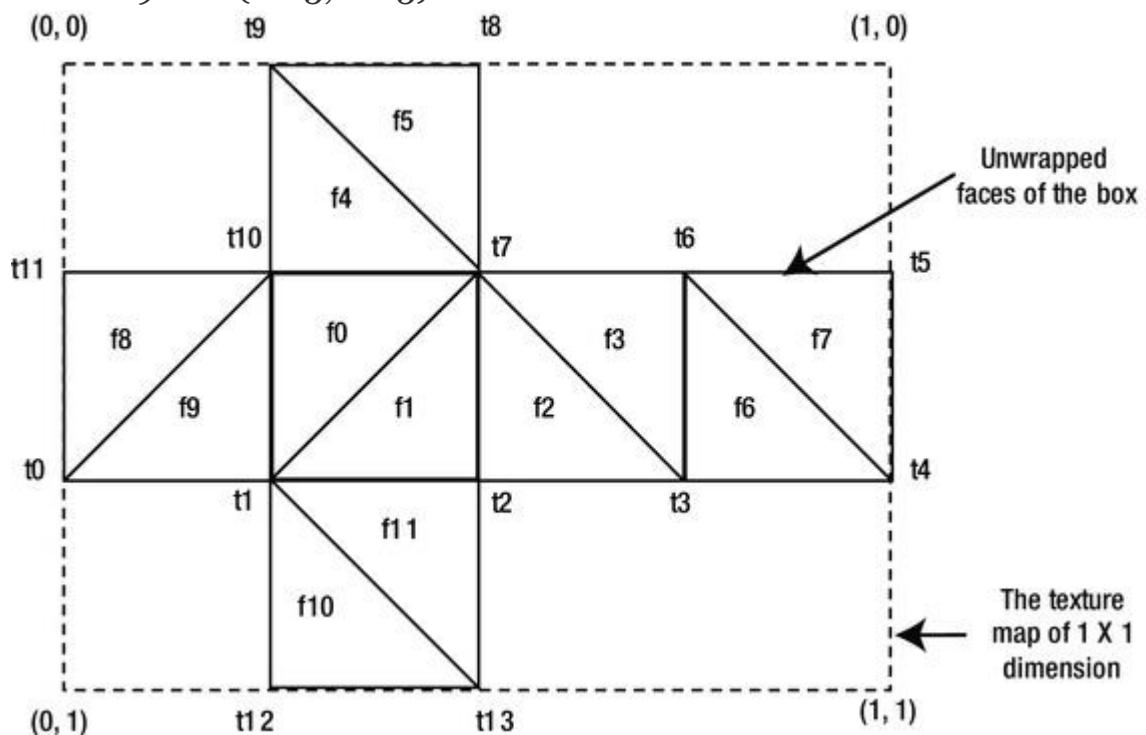
**Tip** It is up to you to decide how you want to map the surface of a 3D object into a 2D plane. For example, in Figure 19-12, you could have also mapped the bottom side of the box into the lower, left, or top of the unit square.

Think of an image that you want to use as the texture for your box. The image will not have the third dimension (z dimension). The image needs to be applied on the surface of the box. JavaFX needs to know how the vertices on the box are mapped to the points on the image. You provide this information in terms of mapping of box vertices to the points on the image.

Now, think of a unit square (a  $1 \times 1$  square) that represents the texture image. Overlay the unit square on the unwrapped faces of the box. The unit square is shown in dotted outline in Figure 19-12. The upper-left corner of the square has the coordinates  $(0, 0)$ ; the lower-left corner has the coordinates  $(0, 1)$ ; the upper-right corner has the coordinates  $(1, 0)$ ; the lower-left corner has the coordinates  $(1, 1)$ .

In Figure 19-12, when you opened the surface of the box to put it onto a 2D plane, some of the vertices had to be split into multiple vertices.

The box has eight vertices. The mapped box into the 2D plane has 14 vertices. The figure shows some of the vertices having the same number as those vertices representing the same vertex in the 3D box. Each vertex mapped into 2D plane (in Figure 19-12) becomes an element in the texture coordinates array. Figure 19-13 shows those 14 texture points; they are numbered as  $t_0$ ,  $t_1$ ,  $t_2$ , and so on. You can number the vertices of the box onto the 2D plane in any order you want. The x and y coordinates of a texture point will be between 0 and 1. The actual mapping of these coordinates to the actual image size is performed by JavaFX. For example,  $(0.25, 0.)$  may be used for the coordinates of the vertex  $t_9$  and  $(0.25, 0.25)$  for the vertex  $t_{10}$ .



**Figure 19-13.** A box surface mapped onto a 2D plane with texture coordinates

You can create the texture coordinates array as shown in the following code. Like the `pointsarray`, following is the pseudo-code. The first array is for understanding the concept and the second array is the actual one that is used in code.

```
// For understanding purpose-only
float[] texCoords = {t0,
                     t1,
                     t2,
                     ...
                     t14};
```

```
// The actual texture coordinates of vertices
float[] texCoords = {x0, y0, // t0
                     x1, y1, // t1
```

```

    x2, y2, // t2
    ...
    x13, y13 // t13
};

```

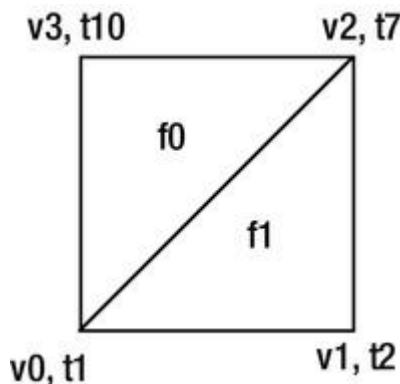
The third piece of information that you need to specify is an array of faces. Note that each triangle has two faces. In our figures, you have shown only the front faces of the triangles. Specifying faces is the most confusing step in creating a `TriangleMesh` object. A face is specified using the `points` array and `texture coordinates` array. You use the indices of the vertices in the `point` array and the indices of the texture points in the `texture coordinates` array to specify a face. A face is specified in using six integers in the following formats:

`iv0, it0, iv1, it1, iv2, it2`

Here,

- `iv0` is the index of the vertex `v0` in the `points` array and `it0` is the index of the point `t0` in the `texture coordinates` array
- `iv1` and `it1` are the indices of the vertex `v1` and point `t1` in the `points` and `texture coordinates` arrays
- `iv2` and `it2` are the indices of the vertex `v2` and point `t2` in the `points` and `texture coordinates` arrays

Figure 19-14 shows only two triangles, which make up the front side of the box.



**Figure 19-14.** Two triangles of the box with their vertices in points and texture coordinates arrays

Figure 19-14 is the superimposition of the figures shown in Figure 19-12 and Figure 19-13. The figure shows the vertex number and their corresponding texture coordinate point number. To specify the `f0` in the `faces` array, you can specify the vertices of the triangle in two ways: counterclockwise and clockwise.

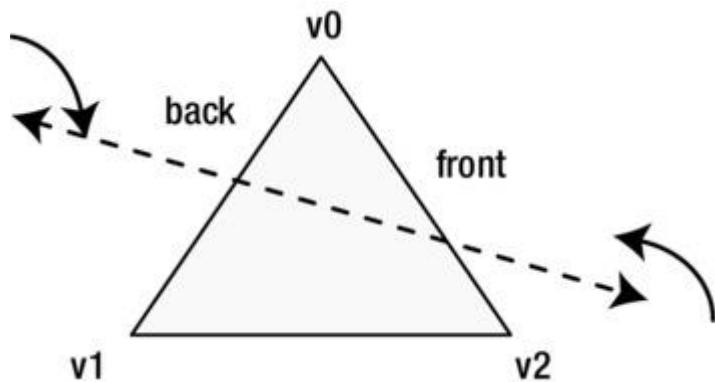
```
ivo, it1, iv2, it7, iv3, it10 (Counterclockwise)
ivo, it1, iv3, it10, iv2, it7 (Clockwise)
```

The starting vertex does not matter in specifying a face. You can start with any vertex and go in a clockwise or a counterclockwise direction. When the vertices for a face are specified in the counterclockwise direction, it is considered the front face. Otherwise, it is considered the back face. The following series of numbers will specify the face f1 in our figure:

```
ivo, it1, iv1, it2, iv2, it7 (Counterclockwise: front-face)
ivo, it1, iv2, it7, iv1, it2 (Clockwise: back-face)
```

To determine whether you are specifying front face or back face, apply the following rules as illustrated in Figure 19-15:

- Draw a line perpendicular to the surface of the triangle going outward.
- Imagine you are looking into the surface by aligning your view along the line.
- Try traversing the vertices in counterclockwise. The sequence of vertices will give you front face. If you traverse the vertices clockwise, the sequence will give you back face.



**Figure 19-15.** Winding order of vertices of a triangle

The following pseudo-code illustrates how to create an `int` array for specifying faces. The `int` values are the array indices from the points and texture coordinates arrays.

```
int[] faces = new int[] {
    ivo, it1, iv2, it7, iv3, it10, // f0: front-face
    ivo, it1, iv3, it10, iv2, it7, // f0: back-face
    ivo, it1, iv1, it2, iv2, it7, // f1: front-face
    ivo, it1, iv2, it7, iv1, it2 // f1: back-face
    ...
};
```

Once you have the points, texture coordinates, and faces arrays, you can construct a `TriangleMesh` object as follows:

```
TriangleMesh mesh = new TriangleMesh();
mesh.getPoints().addAll(points);
mesh.getTexCoords().addAll(texCoords);
mesh.getFaces().addAll(faces);
```

A `TriangleMesh` provides the data for constructing a user-defined 3D object. A `MeshView` object creates the surface for the object with a specified `TriangleMesh`.

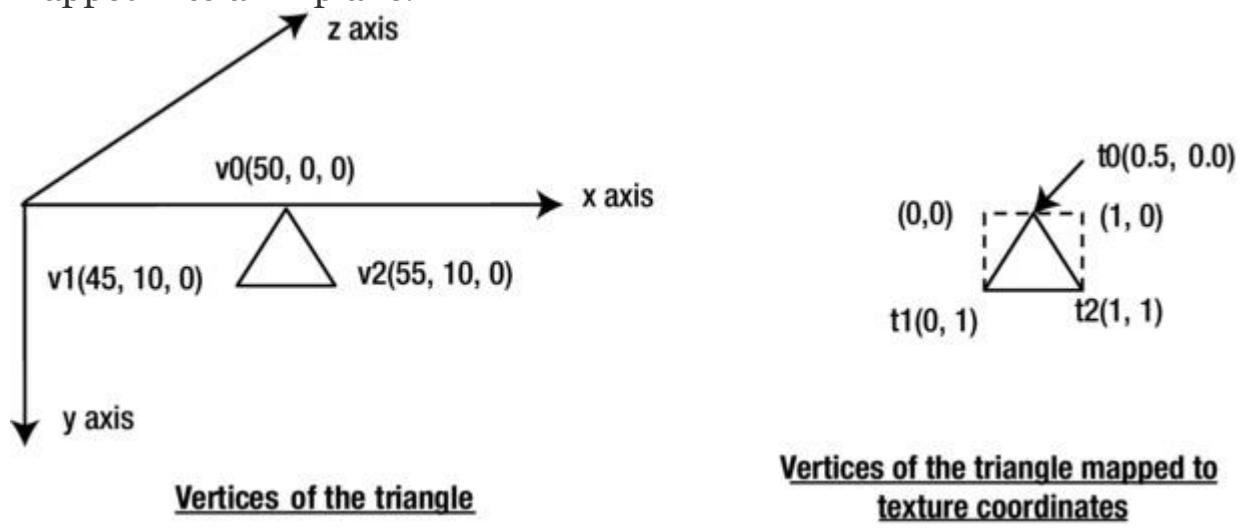
```
// Create a MeshView
MeshView meshView = new MeshView();
meshView.setMesh(mesh);
```

Once you have a `MeshView` object, you need to add it to a scene graph to view it. You can view it the same way you have been viewing the predefined 3D shapes Boxes, Spheres, and Cylinders.

In the next few sections, you will create 3D objects using a `TriangleMesh`. You will start with the simplest 3D object, which is a triangle.

### Creating a 3D Triangle

You may argue that a triangle is a 2D shape, not a 3D shape. It is agreed that a triangle is a 2D shape. You will create a triangle in a 3D space using a `TriangleMesh`. The triangle will have two faces. This example is chosen because it is the simplest shape you can create with a mesh of triangles. In case of a triangle, the mesh consists of only one triangle. Figure 19-16 shows a triangle in the 3D space and its vertices mapped into a 2D plane.



**Figure 19-16.** Vertices of a triangle in the 3D space and mapped onto a 2D plane

The triangle can be created using a mesh of one triangle. Let us create the `points` array for the `TriangleMesh` object.

```
float[] points = {50, 0, 0, // v0 (iv0 = 0)
                 45, 10, 0, // v1 (iv1 = 1)
```

```
    55, 10, 0 // v2 (iv2 = 2)
};
```

The second part of the figure, shown on the right, maps the vertices of the triangle to a unit square. You can create the `textureCoordinates` array as follows:

```
float[] texCoords = {0.5f, 0.5f, // t0 (it0 = 0)
                     0.0f, 1.0f, // t1 (it1 = 1)
                     1.0f, 1.0f // t2 (it2 = 2)
};
```

Using the points and texture coordinates arrays, you can specify the `faces` array as follows:

```
int[] faces = { 0, 0, 2, 2, 1, 1, // iv0, it0, iv2, it2, iv1,
               it1 (front face)
                 0, 0, 1, 1, 2, 2 // iv0, it0, iv1, it1, iv2, it2
back face
};
```

**Listing 19-8** contains the complete program to create a triangle using a `TriangleMesh`. It adds two different lights to light the two faces of the triangle. An animation rotates the camera, so you can view both sides of the triangle in different colors. The `createMeshView()` method has the coordinate values and logic to create the `MeshView`.

### ***Listing 19-8.*** Creating a Triangle Using a `TriangleMesh`

```
// TriangleWithAMesh.java
package com.jdojo.shape3d;

import javafx.animation.Animation;
import javafx.animation.RotateTransition;
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.PerspectiveCamera;
import javafx.scene.PointLight;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.MeshView;
import javafx.scene.shape.TriangleMesh;
import javafx.scene.transform.Rotate;
import javafx.stage.Stage;
import javafx.util.Duration;

public class TriangleWithAMesh extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // Create a MeshView and position ity in the space
        MeshView meshView = this.createMeshView();
```

```
meshView.setTranslateX(250);
meshView.setTranslateY(100);
meshView.setTranslateZ(400);

// Scale the Meshview to make it look bigger
meshView.setScaleX(10.0);
meshView.setScaleY(10.0);
meshView.setScaleZ(10.0);

PerspectiveCamera camera = new
PerspectiveCamera(false);
camera.setTranslateX(100);
camera.setTranslateY(-50);
camera.setTranslateZ(300);

// Add a Rotation animation to the camera
RotateTransition rt = new
RotateTransition(Duration.seconds(2), camera);
rt.setCycleCount(Animation.INDEFINITE);
rt.setFromAngle(-30);
rt.setToAngle(30);
rt.setAutoReverse(true);
rt.setAxis(Rotate.Y_AXIS);
rt.play();

// Front light is red
PointLight redLight = new PointLight();
redLight.setColor(Color.RED);
redLight.setTranslateX(250);
redLight.setTranslateY(150);
redLight.setTranslateZ(300);

// Back light is green
PointLight greenLight = new PointLight();
greenLight.setColor(Color.GREEN);
greenLight.setTranslateX(200);
greenLight.setTranslateY(150);
greenLight.setTranslateZ(450);

Group root = new Group(meshView, redLight,
greenLight);

// Rotate the triangle with its lights to 90 degrees
root.setRotationAxis(Rotate.Y_AXIS);
root.setRotate(90);

Scene scene = new Scene(root, 400, 300, true);
scene.setCamera(camera);
stage.setScene(scene);
stage.setTitle("Creating a Triangle using
a TriangleMesh");
stage.show();
}

public MeshView createMeshView() {
```

```

        float[] points = {50, 0, 0, // v0 (iv0 = 0)
                          45, 10, 0, // v1 (iv1 = 1)
                          55, 10, 0 // v2 (iv2 = 2)
                        };

        float[] texCoords = { 0.5f, 0.5f, // t0 (it0 = 0)
                              0.0f, 1.0f, // t1 (it1 = 1)
                              1.0f, 1.0f // t2 (it2 = 2)
                            };

        int[] faces = {
          0, 0, 2, 2, 1, 1, // iv0, it0, iv2, it2, iv1,
        it1 (front face)
          0, 0, 1, 1, 2, 2 // iv0, it0, iv1, it1, iv2,
        it2 (back face)
        };

        // Create a TriangleMesh
        TriangleMesh mesh = new TriangleMesh();
        mesh.getPoints().addAll(points);
        mesh.getTexCoords().addAll(texCoords);
        mesh.getFaces().addAll(faces);

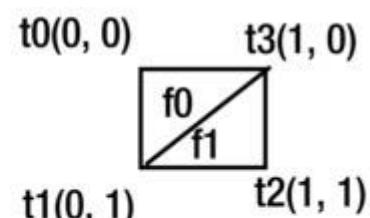
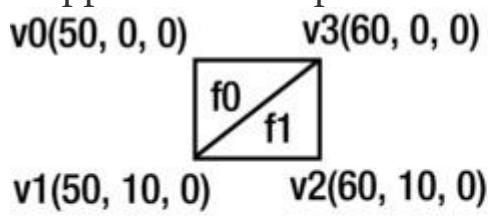
        // Create a MeshView
        MeshView meshView = new MeshView();
        meshView.setMesh(mesh);

        return meshView;
    }
}

```

## Creating a 3D Rectangle

In this section, you will create a rectangle using a mesh of two triangles. This will give us an opportunity to use what you have learned so far. Figure 19-17 shows a rectangle in the 3D space and its vertices mapped into a 2D plane.



**Figure 19-17.** Vertices of a rectangle in the 3D space and mapped into a 2D plane

The rectangle consists of two triangles. Both triangles have two faces. In the figure, I have shown only two faces f0 and f1. The following is the points array for the four vertices of the rectangle.

```
float[] points = {50, 0, 0, // v0 (iv0 = 0)
                  50, 10, 0, // v1 (iv1 = 1)
                  60, 10, 0, // v2 (iv2 = 2)
                  60, 0, 0 // v3 (iv3 = 3)
};
```

The texture coordinate array can be constructed as follows:

```
float[] texCoords = {0.0f, 0.0f, // t0 (it0 = 0)
                     0.0f, 1.0f, // t1 (it1 = 1)
                     1.0f, 1.0f, // t2 (it2 = 2)
                     1.0f, 0.0f // t3 (it3 = 3)
};
```

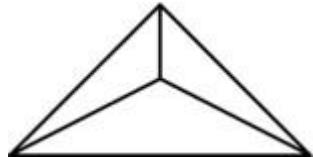
You will specify the four faces as follows:

```
int[] faces =
    { 0, 0, 3, 3, 1, 1, // iv0, it0, iv3, it3, iv1, it1 (f0
front face)
    0, 0, 1, 1, 3, 3, // iv0, it0, iv1, it1, iv3, it3 (f0
back face)
    1, 1, 3, 3, 2, 2, // iv1, it1, iv3, it3, iv2, it2 (f1
front face)
    1, 1, 2, 2, 3, 3 // iv1, it1, iv2, it2, iv3, it3 (f1
back face)
};
```

If you plug the aforementioned three arrays into the `createMeshView()` method in Listing 19-8, you will get a rotating rectangle.

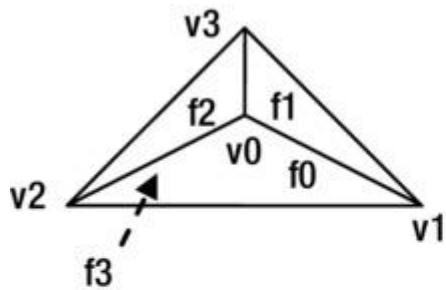
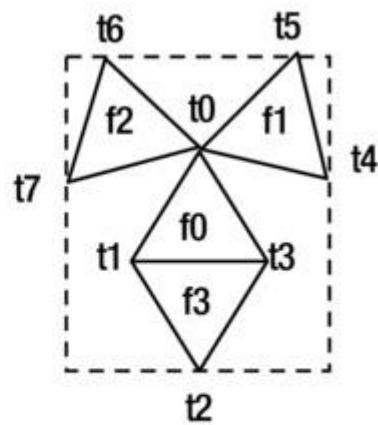
### Creating a Tetrahedron

Now, you are prepared to create a little complex 3D object. You will create a tetrahedron. Figure 19-18 shows the top view of a tetrahedron.



**Figure 19-18.** A tetrahedron

A tetrahedron consists of four triangles. It has four vertices. Three triangles meet at a point. Figure 19-19 shows the two views of the tetrahedron. On the left, you have numbered the four vertices as v0, v1, v2, and v3 and four faces as f0, f1, f2, and f3. Note that the face f3 is the face of the triangle at the base and it is not visible from the top view. The second view has unwrapped the four triangles giving rise to eight vertices on the 2D plane. The dotted rectangle is the unit square into which the eight vertices will be mapped.

**Vertices of the tetrahedron****Vertices of the tetrahedron mapped to texture coordinates*****Figure 19-19.*** Vertices of a tetrahedron in the 3D space and mapped into a 2D plane

You can create the points, faces, and texture coordinates arrays as follows:

```

float[] points = {10, 10, 10, // v0 (iv0 = 0)
                  20, 20, 0, // v1 (iv1 = 1)
                  0, 20, 0, // v2 (iv2 = 2)
                  10, 20, 20 // v3 (iv3 = 3)
                };

float[] texCoords = {
    0.50f, 0.33f, // t0 (it0 = 0)
    0.25f, 0.75f, // t1 (it1 = 1)
    0.50f, 1.00f, // t2 (it2 = 2)
    0.66f, 0.66f, // t3 (it3 = 3)
    1.00f, 0.35f, // t4 (it4 = 4)
    0.90f, 0.00f, // t5 (it5 = 5)
    0.10f, 0.00f, // t6 (it6 = 6)
    0.00f, 0.35f // t7 (it7 = 7)
};

int[] faces = {
    0, 0, 2, 1, 1, 3, // f0 front-face
    0, 0, 1, 3, 2, 1, // f0 back-face
    0, 0, 1, 4, 3, 5, // f1 front-face
    0, 0, 3, 5, 1, 4, // f1 back-face
    0, 0, 3, 6, 2, 7, // f2 front-face
    0, 0, 2, 7, 3, 6, // f2 back-face
    1, 3, 3, 2, 2, 1, // f3 front-face
    1, 3, 2, 1, 3, 2 // f3 back-face
};

```

Listing 19-9 contains a complete program to show how to construct a tetrahedron using a TriangleMesh. The tetrahedron is rotated along y-axis, so you can view two of its vertical faces. Figure 19-20 shows the window with the tetrahedron.

**Listing 19-9.** Creating a Tetrahedron Using a TriangleMesh

```
// Tetrahedron.java
package com.jdojo.shape3d;

import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.PerspectiveCamera;
import javafx.scene.PointLight;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.MeshView;
import javafx.scene.shape.TriangleMesh;
import javafx.scene.transform.Rotate;
import javafx.stage.Stage;

public class Tetrahedron extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        MeshView meshView = this.createMeshView();
        meshView.setTranslateX(250);
        meshView.setTranslateY(50);
        meshView.setTranslateZ(400);

        meshView.setScaleX(10.0);
        meshView.setScaleY(20.0);
        meshView.setScaleZ(10.0);

        PerspectiveCamera camera = new
PerspectiveCamera(false);
        camera.setTranslateX(100);
        camera.setTranslateY(0);
        camera.setTranslateZ(100);

        PointLight redLight = new PointLight();
        redLight.setColor(Color.RED);
        redLight.setTranslateX(250);
        redLight.setTranslateY(-100);
        redLight.setTranslateZ(250);

        Group root = new Group(meshView, redLight);
        root.setRotationAxis(Rotate.Y_AXIS);
        root.setRotate(45);

        Scene scene = new Scene(root, 200, 150, true);
        scene.setCamera(camera);
        stage.setScene(scene);
        stage.setTitle("A Tetrahedron using
a TriangleMesh");
        stage.show();
    }
}
```

```
public MeshView createMeshView() {
    float[] points = {10, 10, 10, // v0 (iv0 = 0)
                      20, 20, 0, // v1 (iv1 = 1)
                      0, 20, 0, // v2 (iv2 = 2)
                      10, 20, 20 // v3 (iv3 = 3)
                     };

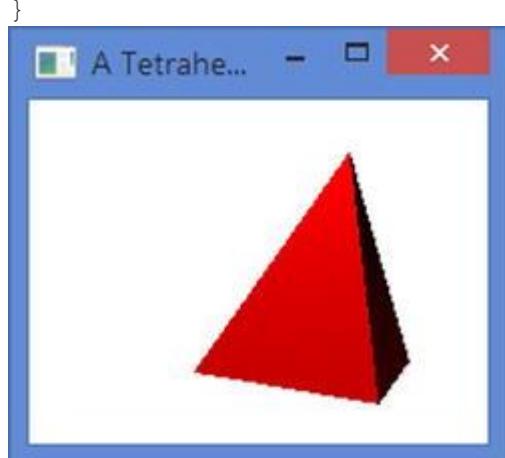
    float[] texCoords = {
        0.50f, 0.33f, // t0 (it0 = 0)
        0.25f, 0.75f, // t1 (it1 = 1)
        0.50f, 1.00f, // t2 (it2 = 2)
        0.66f, 0.66f, // t3 (it3 = 3)
        1.00f, 0.35f, // t4 (it4 = 4)
        0.90f, 0.00f, // t5 (it5 = 5)
        0.10f, 0.00f, // t6 (it6 = 6)
        0.00f, 0.35f // t7 (it7 = 7)
    };

    int[] faces = {
        0, 0, 2, 1, 1, 3, // f0 front-face
        0, 0, 1, 3, 2, 1, // f0 back-face
        0, 0, 1, 4, 3, 5, // f1 front-face
        0, 0, 3, 5, 1, 4, // f1 back-face
        0, 0, 3, 6, 2, 7, // f2 front-face
        0, 0, 2, 7, 3, 6, // f2 back-face
        1, 3, 3, 2, 2, 1, // f3 front-face
        1, 3, 2, 1, 3, 2, // f3 back-face
    };
}

TriangleMesh mesh = new TriangleMesh();
mesh.getPoints().addAll(points);
mesh.getTexCoords().addAll(texCoords);
mesh.getFaces().addAll(faces);

MeshView meshView = new MeshView();
meshView.setMesh(mesh);

return meshView;
}
```



**Figure 19-20.** A tetrahedron using a *TriangleMesh*

## Summary

Any shape, drawn in a three-dimensional space, having three dimensions (length, width, and depth), is known as a 3D shape such as cubes, spheres, pyramids, and so on. JavaFX 8 provides 3D shapes as nodes. JavaFX 8 offers two types of 3D shapes: predefined shapes and user-defined shapes.

Box, sphere, and cylinder are three predefined 3D shapes that you can readily use in your JavaFX applications. You can create any type of 3D shapes using a triangle mesh. The `Box`, `Sphere`, and `Cylinder` classes represent the three predefined shapes. The `MeshView` class represents a user-defined 3D shape in a scene. The 3D shape classes are in the `javafx.scene.shape` package.

JavaFX 3D support is a conditional feature. If it is not supported on your platform, you get a warning message on the console when you run a program that attempts to use 3D features. The method `Platform.isSupported(ConditionalFeature.SCENE3D)` returns `true` if 3D is supported on your platform.

When dealing with 3D objects in JavaFX, you would like the object closer to you to overlap the object farther from you. In JavaFX, by default, nodes are rendered in the order they are added to the scene graph. In order for 3D shapes to appear as they would appear in the real world, you need to specify two things. First, when you create a `Scene` object, specify that it needs to have a depth buffer, and second, specify that the nodes' z coordinate values should be used when they are rendered.

Cameras are used to render the scene. Cameras in JavaFX are nodes. They can be added to the scene graph and positioned like other nodes. Perspective camera and parallel camera are two types of cameras used in JavaFX and they are represented by the `PerspectiveCamera` and `ParallelCamera` classes. A perspective camera defines the viewing volume for a perspective projection, which is a truncated right pyramid. The camera projects the objects contained within the near and far clipping planes onto the projection plane. Therefore, any objects outside the clipping planes are not visible. A parallel camera specifies the viewing volume for a parallel projection, which is a rectangular box.

Similar to the real world, you need a light source to view the 3D objects in a scene. An instance of the abstract base class `LightBase` represents a light source. Its two concrete

subclasses, `AmbientLight` and `PointLight` represent an ambient light and a point light.

A scene can use only one camera. Sometimes, you may want to view different parts of a scene using multiple cameras. JavaFX 8 introduces the concept as subscenes. A subscene is a container for a scene graph. It can have its own width, height, fill color, depth buffer, antialiasing flag, and camera. An instance of the `SubScene` class represents a subscene. The `SubScene` inherits from the `Node` class.

The next chapter will discuss how to apply different types of effects to nodes in a scene graph.

## CHAPTER 20



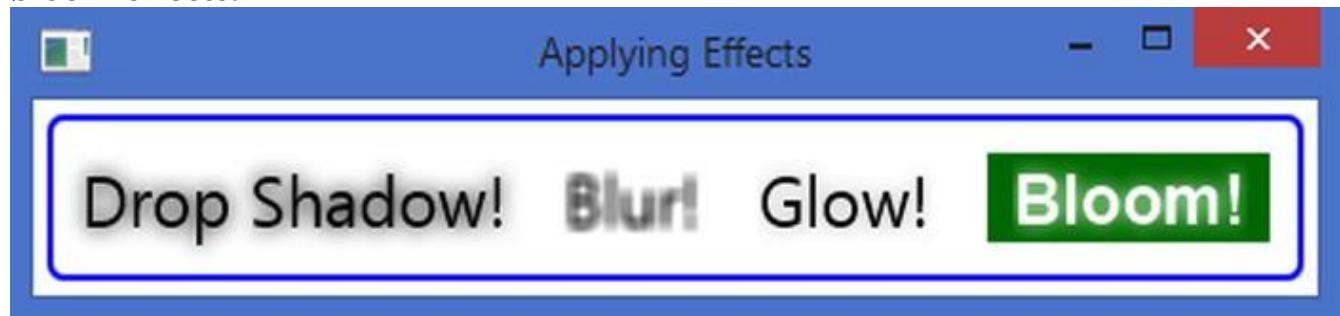
### Applying Effects

In this chapter, you will learn:

- What an effect is
- How to chain effects
- What different types of effects are
- How to use perspective transformation effect

#### What Is an Effect?

An effect is a filter that accepts one or more graphical inputs, applies an algorithm on the inputs, and produces an output. Typically, effects are applied to nodes to create visually appealing user interfaces. Examples of effects are shadow, blur, warp, glow, reflection, blending, different types of lighting, among others. The JavaFX library provides several effect-related classes. Effects are conditional features. They are applied to nodes and will be ignored if they are not available on a platform. Figure 20-1 shows four `Text` nodes using the drop shadow, blur, glow, and bloom effects.



**Figure 20-1.** Text nodes with different effects

The `Node` class contains an `effect` property that specifies the effect applied to the node. By default, it is `null`. The following snippet of code applies a drop shadow effect to a `Text` node:

```
Text t1 = new Text("Drop Shadow");
t1.setFont(Font.font(24));
t1.setEffect(new DropShadow());
```

An instance of the `Effect` class represents an effect.

The `Effect` class is the abstract base for all effect classes. All effect classes are included in the `javafx.scene.effect` package.

The program in Listing 20-1 creates `Text` nodes and applies effects to them. These nodes are the ones shown in Figure 20-1. I will explain the different types of effects and their usages in subsequent sections.

### ***Listing 20-1.*** Applying Effects to Nodes

```
// EffectTest.java
package com.jdojo.effect;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.effect.Bloom;
import javafx.scene.effect.BoxBlur;
import javafx.scene.effect.DropShadow;
import javafx.scene.effect.Glow;
import javafx.scene.layout.HBox;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class EffectTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Text t1 = new Text("Drop Shadow!");
        t1.setFont(Font.font(24));
        t1.setEffect(new DropShadow());

        Text t2 = new Text("Blur!");
        t2.setFont(Font.font(24));
        t2.setEffect(new BoxBlur());

        Text t3 = new Text("Glow!");
        t3.setFont(Font.font(24));
        t3.setEffect(new Glow());

        Text t4 = new Text("Bloom!");
        t4.setFont(Font.font("Arial", FontWeight.BOLD, 24));
        t4.setFill(Color.WHITE);
        t4.setEffect(new Bloom(0.10));

        // Stack the Text node with bloom effect over
        // a Rectangle
        Rectangle rect = new Rectangle(100, 30,
        Color.GREEN);
        StackPane spane = new StackPane(rect, t4);
```

```

        HBox root = new HBox(t1, t2, t3, spane);
        root.setSpacing(20);
        root.setStyle("-fx-padding: 10;" +
                      "-fx-border-style: solid inside;" +
                      "-fx-border-width: 2;" +
                      "-fx-border-insets: 5;" +
                      "-fx-border-radius: 5;" +
                      "-fx-border-color: blue;");

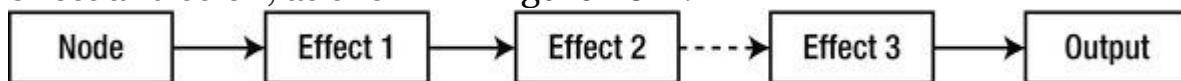
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Applying Effects");
        stage.show();
    }
}

```

**Tip** An effect applied to a Group is applied to all its children. It is also possible to chain multiple effects where the output of one effect becomes the input for the next effect in the chain. The layout bounds of a node are not affected by the effects applied to it. However, the local bounds and bounds in parent are affected by the effects.

## Chaining Effects

Some effects can be chained with other effects when they are applied in sequence. The output of the first effect becomes the input for the second effect and so on, as shown in Figure 20-2.



**Figure 20-2.** A chain of effects applied on a node

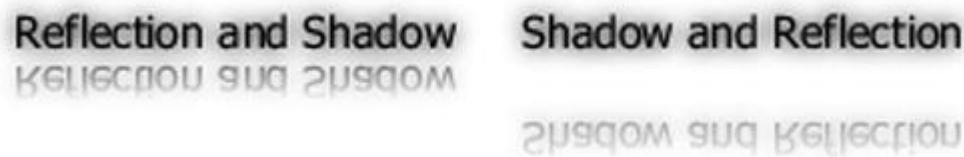
Effect classes that allow chaining contain an `input` property to specify the effect that precedes it. If the `input` is `null`, the effect is applied to the node on which this effect is set instead of being applied to the preceding input effect. By default, the `input` is `null`. The following snippet of code creates two chains of effects on `Text` nodes, as shown in Figure 20-3:

```

// Effect Chain: Text >> Reflection >> Shadow
DropShadow dsEffect = new DropShadow();
dsEffect.setInput(new Reflection());
Text t1 = new Text("Reflection and Shadow");
t1.setEffect(dsEffect);

// Effect Chain: Text >> Shadow >> Reflection
Reflection reflection = new Reflection();
reflection.setInput(new DropShadow());
Text t2 = new Text("Shadow and Reflection");
t2.setEffect(reflection);

```



**Figure 20-3.** Chaining a `DropShadow` effect with a `Reflection` effect

In Figure 20-3, a `Reflection` effect followed by a `DropShadow` is applied to the text on the left; a `DropShadow` followed by a `Reflection` effect is applied to the text on the right. Notice the sequence of effects makes a difference in the output. The second chain of effects produces a taller output as the reflection also includes the shadow.

If an effect allows chaining, it will have an `input` property. In subsequent sections, I will list the `input` property for the effect classes, but not discuss it.

## Shadowing Effects

A shadowing effect draws a shadow and applies it to an input. JavaFX supports three types of shadowing effects:

- `DropShadow`
- `InnerShadow`
- `Shadow`

### The `DropShadow` Effect

The `DropShadow` effect draws a shadow (a blurred image) behind the input, so the input seems to be raised. It gives the input a 3D look. The input can be a node or an effect in a chain of effects.

An instance of the `DropShadow` class represents a `DropShadow` effect. The size, location, color, and quality of the effect are controlled by several properties of the `DropShadow` class:

- `offsetX`
- `offsetY`
- `color`
- `blurType`
- `radius`
- `spread`
- `width`
- `height`
- `input`

The `DropShadow` class contains several constructors that let you specify the initial values for the properties:

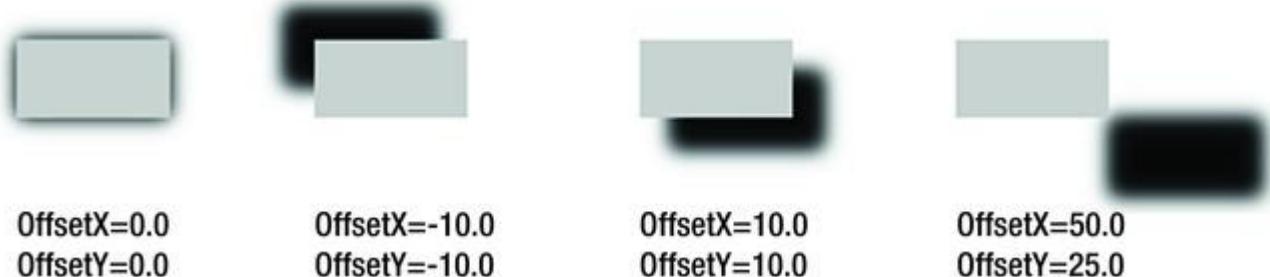
- `DropShadow()`
- `DropShadow(BlurType blurType, Color color, double radius, double spread, double offsetX, double offsetY)`
- `DropShadow(double radius, Color color)`
- `DropShadow(double radius, double offsetX, double offsetY, Color color)`

The `offsetX` and `offsetY` properties control the position of the shadow in pixels relative to the input. By default, their values are zero. The positive values of `offsetX` and `offsetY` move the shadow in the positive x axis and y axis directions, respectively. The negative values move the shadow in the reverse directions.

The following snippet of code creates a `DropShadow` object with the `offsetX` and `offsetY` of 10px. The third rectangle from the left in Figure 20-4 shows the rectangle with the effect using the same rectangle with a `DropShadow` effect and different x and y offsets. For the fourth from the left rectangle, the shadow is positioned at the lower right corner of the rectangle as the rectangle size (50, 25) matches the offsets (50, 25).

```
DropShadow dsEffect = new DropShadow();
dsEffect.setOffsetX(10);
dsEffect.setOffsetY(10);

Rectangle rect = new Rectangle(50, 25, Color.LIGHTGRAY);
rect.setEffect(dsEffect);
```



**Figure 20-4.** Effects of the `offsetX` and `offsetY` properties on a `DropShadow` effect

The `color` property specifies the color of the shadow. By default, it is `Color.BLACK`. The following code would set the color to red:

```
DropShadow dsEffect = new DropShadow();
dsEffect.setColor(Color.RED);
```

The blurring in the shadow can be achieved using different algorithms. The `blurType` property specifies the type of blurring algorithm for the shadow. Its value is one of the following constants of the `BlurType` enum:

- `ONE_PASS_BOX`
- `TWO_PASS_BOX`
- `THREE_PASS_BOX`
- `GAUSSIAN`

The `ONE_PASS_BOX` uses a single pass of the box filter to blur the shadow. The `TWO_PASS_BOX` uses two passes of the box filter to blur the shadow. The `THREE_PASS_BOX` uses three passes of the box filter to blur the shadow. The `GAUSSIAN` uses a Gaussian blur kernel to blur the shadow. The blur quality of the shadow is the least in `ONE_PASS_BOX` and the best in `GAUSSIAN`. The default is `THREE_PASS_BOX`, which is very close to `GAUSSIAN` in quality. The following snippet of code sets the `GAUSSIAN` blur type:

```
DropShadow dsEffect = new DropShadow();
dsEffect.setBlurType(BlurType.GAUSSIAN);
```

The `radius` property specifies the distance the shadow is spread on each side of the source pixel. If the radius is zero, the shadow has sharp edges. Its value can be between 0 and 127. The default value is 10. The blurring outside the shadow region is achieved by blending the shadow color and the background color. The blur color fades out over the radius distance from the edges.

Figure 20-5 shows a rectangle twice with a `DropShadow` effect. The one on the left uses the `radius` of 0.0, which results in sharp edges of the shadow. The one on the right uses the default radius of 10.0 that spreads the shadow 10px around the edges. The following snippet of code produces the first rectangle in the figure that has sharp edges of the shadow:

```
DropShadow dsEffect = new DropShadow();
dsEffect.setOffsetX(10);
dsEffect.setOffsetY(10);
dsEffect.setRadius(0);
```

```
Rectangle rect = new Rectangle(50, 25, Color.LIGHTGRAY);
rect.setEffect(dsEffect);
```



`radius=0.0`



`radius=10.0`

**Figure 20-5.** Effects of the radius property of a DropShadow effect

The spread property specifies the portion of the radius, which has the same color as the shadow. The color for the remaining portion of the radius is determined by the blur algorithm. Its value is between 0.0 and 1.0. The default is 0.0.

Suppose you have a DropShadow with a radius 10.0 and a spread value of 0.60 and the shadow color is black. In this case, the blur color will be black up to 6px around the source pixel. It will start fading out from the seventh pixel to the tenth pixel. If you specify the spread value as 1.0, there would be no blurring of the shadow. Figure 20-6 shows three rectangles with a DropShadow using a radius of 10.0. The three DropShadow effects use different spread values. The spread of 0.0 blurs fully along the radius. The spread of 0.50 spreads the shadow color in the first half of the radius and blurs the second half. The spread of 1.0 spreads the shadow color fully along the radius and there is no blurring.

The following snippet of code produces the middle rectangle in Figure 20-6:

```
DropShadow dsEfefct = new DropShadow();
dsEfefct.setOffsetX(10);
dsEfefct.setOffsetY(10);
dsEfefct.setRadius(10);
dsEfefct.setSpread(.50);

Rectangle rect = new Rectangle(50, 25, Color.LIGHTGRAY);
rect.setEffect(dsEfefct);
```



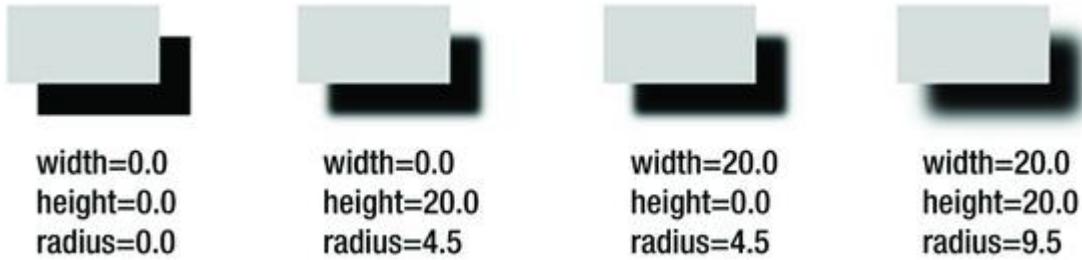
**Figure 20-6.** Effects of the spread property of a DropShadow effect

The width and height properties specify the horizontal and vertical distances, respectively, from the source pixel up to where the shadow color is spread. Their values are between 0 and 255. Setting their values is equivalent to setting the radius property, so they are equal to  $(2 * \text{radius} + 1)$ . Their default value is 21.0. When you change the radius, the width and height properties are adjusted using the formula if they are not bound. However, setting the width and height changes the radius value, so the average of the width and height is equal to  $(2 * \text{radius} + 1)$ . Figure 20-7 shows four rectangles with a DropShadow effects. Their width and height properties were set as shown under each rectangle. Their radius properties were adjusted

automatically. The fourth from the left rectangle was produced using the following snippet of code:

```
DropShadow dsEffect = new DropShadow();
dsEffect.setOffsetX(10);
dsEffect.setOffsetY(10);
dsEffect.setWidth(20);
dsEffect.setHeight(20);

Rectangle rect = new Rectangle(50, 25, Color.LIGHTGRAY);
rect.setEffect(dsEffect);
```



**Figure 20-7.** Effects of setting width and height of a DropShadow

The program in Listing 20-2 lets you experiment with properties of the DropShadow effect. It displays a window as shown in Figure 20-8. Change the properties to see their effects in action.

### **Listing 20-2.** Experimenting with DropShadow Properties

```
// DropShadowTest.java
package com.jdojo.effect;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ColorPicker;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.control.Slider;
import javafx.scene.effect.BlurType;
import javafx.scene.effect.DropShadow;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

public class DropShadowTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Rectangle rect = new Rectangle(100, 50, Color.GRAY);
```

```

DropShadow dsEffect = new DropShadow();
rect.setEffect(dsEffect);

GridPane controllerPane
= this.getControllerPane(dsEffect);
BorderPane root = new BorderPane();
root.setCenter(rect);
root.setBottom(controllerPane);
root.setStyle("-fx-padding: 10;" +
             "-fx-border-style: solid inside;" +
             "-fx-border-width: 2;" +
             "-fx-border-insets: 5;" +
             "-fx-border-radius: 5;" +
             "-fx-border-color: blue;");

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("Experimenting with DropShadow
Effect");
stage.show();
}

private GridPane getControllerPane(final DropShadow
dsEffect) {
    Slider offsetXSlider = new Slider(-200, 200, 0);
    dsEffect.offsetXProperty().bind(offsetXSlider.valueP
roperty());

    Slider offsetYSlider = new Slider(-200, 200, 0);
    dsEffect.offsetYProperty().bind(offsetYSlider.valueP
roperty());

    Slider radiusSlider = new Slider(0, 127, 10);
    dsEffect.radiusProperty().bind(radiusSlider.valuePro
perty());

    Slider spreadSlider = new Slider(0.0, 1.0, 0);
    dsEffect.spreadProperty().bind(spreadSlider.valuePro
perty());

    ColorPicker colorPicker = new
ColorPicker(Color.BLACK);
    dsEffect.colorProperty().bind(colorPicker.valuePrope
rty());

    ComboBox<BlurType> blurTypeList = new ComboBox<>();
    blurTypeList.setValue(dsEffect.getBlurType());
    blurTypeList.getItems().addAll(BlurType.ONE_PASS_BOX
    ,
        BlurType.TWO_PASS_BOX,
        BlurType.THREE_PASS_BOX,
        BlurType.GAUSSIAN);
    dsEffect.blurTypeProperty().bind(blurTypeList.valueP
roperty());
}

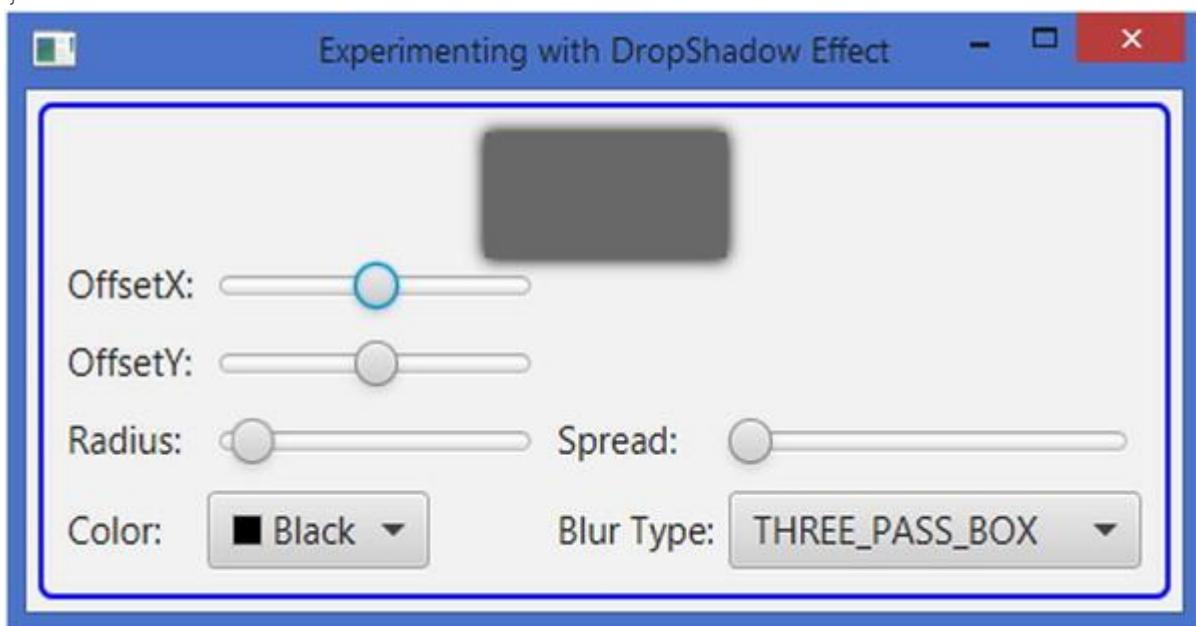
```

```

        GridPane pane = new GridPane();
        pane.setHgap(5);
        pane.setVgap(10);
        pane.addRow(0, new Label("OffsetX:"),
offsetXSlider);
        pane.addRow(1, new Label("OffsetY:"),
offsetYSlider);
        pane.addRow(2, new Label("Radius:"), radiusSlider,
new Label("Spread:"), spreadSlider);
        pane.addRow(3, new Label("Color:"), colorPicker,
new Label("Blur Type:"), blurTypeList);

        return pane;
    }
}

```



**Figure 20-8.** A window that allows you to change the properties of a `DropShadow` effect at runtime

### The `InnerShadow` Effect

The `InnerShadow` effect works very similar to the `DropShadow` effect. It draws a shadow (a blurred image) of an input inside the edges of the input, so the input seems to have depth or a 3D look. The input can be a node or an effect in a chain of effects.

An instance of the `InnerShadow` class represents an `InnerShadow` effect. The size, location, color, and quality of the effect are controlled by several properties of the `InnerShadow` class:

- `offsetX`
- `offsetY`
- `color`

- blurType
- radius
- choke
- width
- height
- input

The number of properties of the `InnerShadow` class is equal to that for the `DropShadow` class. The `spread` property in the `DropShadow` class is replaced by the `choke` property in the `InnerShadow` class, which works similar to the `spread` property in the `DropShadow` class. Please refer to the previous section “The *DropShadow* Effect” for a detailed description and examples of these properties.

The `DropShadow` class contains several constructors that let you specify the initial values for the properties:

- `InnerShadow()`
- `InnerShadow(BlurType blurType, Color color, double radius, double choke, double offsetX, double offsetY)`
- `InnerShadow(double radius, Color color)`
- `InnerShadow(double radius, double offsetX, double offsetY, Color color)`

The program in Listing 20-3 creates a `Text` node and two `Rectangle` nodes. An `InnerShadow` is applied to all three nodes. Figure 20-9 shows the results for these nodes. Notice that the shadow is not spread outside the edges of the nodes. You need to set the `offsetX` and `offsetY` properties to see a noticeable effect.

### ***Listing 20-3.*** Using `InnerShadow` Class

```
// InnerShadowTest.java
package com.jdojo.effect;

import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.effect.InnerShadow;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.shape.Shape;
import javafx.scene.text.Font;
```

```

import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class InnerShadowTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        InnerShadow is1 = new InnerShadow();
        is1.setOffsetX(3);
        is1.setOffsetY(6);

        Text t1 = new Text("Inner Shadow");
        t1.setEffect(is1);
        t1.setFill(Color.RED);
        t1.setFont(Font.font(null, FontWeight.BOLD, 36));

        InnerShadow is2 = new InnerShadow();
        is2.setOffsetX(3);
        is2.setOffsetY(3);
        is2.setColor(Color.GRAY);
        Rectangle rect1 = new Rectangle(100, 50,
Color.LIGHTGRAY);
        rect1.setEffect(is2);

        InnerShadow is3 = new InnerShadow();
        is3.setOffsetX(-3);
        is3.setOffsetY(-3);
        is3.setColor(Color.GRAY);
        Rectangle rect2 = new Rectangle(100, 50,
Color.LIGHTGRAY);
        rect2.setEffect(is3);

        HBox root = new HBox(wrap(t1, is1), wrap(rect1,
is2), wrap(rect2, is3));
        root.setSpacing(10);
        root.setStyle("-fx-padding: 10;" +
"-fx-border-style: solid inside;" +
"-fx-border-width: 2;" +
"-fx-border-insets: 5;" +
"-fx-border-radius: 5;" +
"-fx-border-color: blue;");
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Applying InnerShadow Effect");
        stage.show();
    }

    private VBox wrap(Shape s, InnerShadow in) {
        Text t = new Text ("offsetX=" + in.getOffsetX()
+ "\n" +

```

```

        "offsetY=" + in.getOffsetY());
t.setFont(Font.font(10));

VBox box = new VBox(10, s, t);
box.setAlignment(Pos.CENTER);
return box;
}
}

```



**Figure 20-9.** A Text and two Rectangle nodes using InnerShadow effects

### The Shadow Effect

The `Shadow` effect creates a shadow with blurry edges of its input. Unlike `DropShadow` and `InnerShadow`, it modifies the original input itself to convert it into a shadow. Typically, a `ShadowEffect` is combined with the original input to create a higher-level shadowing effect:

- You can apply a `Shadow` effect with a light color to a node and superimpose it on a duplicate of the original node to create a glow effect.
- You can create a `Shadow` effect with a dark color and place it behind the original node to create a `DropShadow` effect.

An instance of the `Shadow` class represents a `Shadow` effect. The size, color, and quality of the effect are controlled by several properties of the `Shadow` class:

- `color`
- `blurType`
- `radius`
- `width`
- `height`
- `input`

These properties work the same way they work in the DropShadow. Please refer to the section “The *DropShadow* Effect” for a detailed description and examples of these properties.

The Shadow class contains several constructors that let you specify the initial values for the properties:

- `Shadow()`
- `Shadow(BlurType blurType, Color color, double radius)`
- `Shadow(double radius, Color color)`

The program in Listing 20-4 demonstrates how to use the `Shadow` effect. It creates three `Text` nodes. A shadow is applied to all three nodes. The output of the first shadow is displayed. The output of the second shadow is superimposed on the original node to achieve a glow effect. The output of the third shadow is placed behind its original node to achieve a `DropShadow` effect. Figure 20-10 shows these three nodes.

#### ***Listing 20-4.*** Using a `Shadow` Effect and Creating High-Level Effects

```
// ShadowTest.java
package com.jdojo.effect;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.effect.Shadow;
import javafx.scene.layout.HBox;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class ShadowTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // Create a Shadow of a Text node
        Text t1 = new Text("Shadow");
        t1.setFont(Font.font(36));
        t1.setEffect(new Shadow());

        // Create a Glow effect using a Shadow
        Text t2Original = new Text("Glow");
        t2Original.setEffect(new Shadow(GlowType.SUPERIMPOSED, Color.GOLD, 10, 0.5));
```

```

t2Original.setFont(Font.font(36));
Text t2 = new Text("Glow");
t2.setFont(Font.font(36));
Shadow s2 = new Shadow();
s2.setColor(Color.YELLOW);
t2.setEffect(s2);
StackPane glow = new StackPane(t2Original, t2);

// Create a DropShadow effect using a Shadow
Text t3Original = new Text("DropShadow");
t3Original.setFont(Font.font(36));
Text t3 = new Text("DropShadow");
t3.setFont(Font.font(36));
Shadow s3 = new Shadow();
t3.setEffect(s3);
StackPane dropShadow = new StackPane(t3,
t3Original);

HBox root = new HBox(t1, glow, dropShadow);
root.setSpacing(20);
root.setStyle("-fx-padding: 10;" +
"-fx-border-style: solid inside;" +
"-fx-border-width: 2;" +
"-fx-border-insets: 5;" +
"-fx-border-radius: 5;" +
"-fx-border-color: blue;");

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("Using Shadow Effect");
stage.show();
}
}

```



**Figure 20-10.** Applying a shadow to a Text node and creating Glow and DropShadow effects

## Blurring Effects

A blurring effect produces a blurred version of an input. JavaFX lets you apply different types of blurring effects, which differ in the algorithms used to create these effect.

### The BoxBlur Effect

The `BoxBlur` effect uses a box filter kernel to produce a blurring effect. An instance of the `BoxBlur` class represents a `BoxBlur` effect. The size and quality of the effect can be configured using these properties of the class:

- `width`
- `height`
- `iterations`
- `input`

The `width` and `height` properties specify the horizontal and vertical size of the effect, respectively. Imagine a box defined by the `width` and `height` centered on a pixel of the input. The color information of the pixel is spread within the box during the blurring process. The values of these properties are between 5.0 and 255.0. The default values are 5.0. A value of less than or equal to 1.0 does not produce the blurring effect in the corresponding direction.

The `iterations` property specifies the number of times the blurring effect is applied. A higher value produces a better quality blur. Its value can be between 0 and 3. The default is 1. The value of 3 produces the blur quality comparable to the Gaussian blur, discussed in the next section. The value of zero produces no blur at all.

The `BoxBlur` class contains two constructors:

- `BoxBlur()`
- `BoxBlur(double width, double height, int iterations)`

The no-args constructor creates a `BoxBlur` object with the `width` and `height` of 5.0 pixels and `iterations` of 1. The other constructor lets you specify the initial value for the `width`, `height`, and `iterations` properties, as in the following section of code:

```
// Create a BoxBlur with defaults: width=5.0, height=5.0,
iterations=1
BoxBlur bb1 = new BoxBlur();

// Create a BoxBlur with width=10.0, height=10.0, iterations=3
BoxBlur bb2 = new BoxBlur(10, 10, 3);
```

The following snippet of code creates four `Text` nodes and applies `BoxBlur` effects of various qualities. Figure 20-11 show the results of these `Text` nodes. Notice that the last `Text` node does not have any blur effect as the `iterations` property is set to zero.

```
Text t1 = new Text("Box Blur");
t1.setFont(Font.font(24));
```

```
t1.setEffect(new BoxBlur(5, 10, 1));

Text t2 = new Text("Box Blur");
t2.setFont(Font.font(24));
t2.setEffect(new BoxBlur(10, 5, 2));

Text t3 = new Text("Box Blur");
t3.setFont(Font.font(24));
t3.setEffect(new BoxBlur(5, 5, 3));

Text t4 = new Text("Box Blur");
t4.setFont(Font.font(24));
t4.setEffect(new BoxBlur(5, 5, 0)); // Zero iterations = No
blurring
```



**Figure 20-11.** Text nodes with `BoxBlur` effects of varying qualities

### The `GaussianBlur` Effect

The `GaussianBlur` effect uses a Gaussian convolution kernel to produce a blurring effect. An instance of the `GaussianBlur` class represents a `GaussianBlur` effect. The effect can be configured using two properties of the class:

- `radius`
- `input`

The `radius` property controls the distribution of the blur in pixels from the source pixel. The greater this value, the more the blur effect. Its value can be between 0.0 and 63.0. The default value is 10.0. A radius of zero pixels produces no blur effect.

The `GaussianBlur` class contains two constructors:

- `GaussianBlur()`
- `GaussianBlur(double radius)`

The no-args constructor creates a `GaussianBlur` object with a default radius of 10.0px. The other constructor lets you specify the initial value for the radius, as in the following code:

```
// Create a GaussianBlur with a 10.0 pixels radius
GaussianBlur gb1 = new GaussianBlur();
```

```
// Create a GaussianBlur with a 20.0 pixels radius
GaussianBlur gb2 = new GaussianBlur(20);
```

The following snippet of code creates four `Text` nodes and applies `GaussianBlur` effects of different radius values. Figure 20-12 show the results of these `Text` nodes. Notice that the last `Text` node does not have any blur effect as the `radius` property is set to zero.

```
Text t1 = new Text("Gaussian Blur");
t1.setFont(Font.font(24));
t1.setEffect(new GaussianBlur(5));

Text t2 = new Text("Gaussian Blur");
t2.setFont(Font.font(24));
t2.setEffect(new GaussianBlur(10));

Text t3 = new Text("Gaussian Blur");
t3.setFont(Font.font(24));
t3.setEffect(new GaussianBlur(15));

Text t4 = new Text("Gaussian Blur");
t4.setFont(Font.font(24));
t4.setEffect(new GaussianBlur(0)); // radius = 0 means no blur
```



**Figure 20-12.** Text nodes with `GaussianBlur` effects of varying sizes

### The `MotionBlur` Effect

The `MotionBlur` effect produces a blurring effect by motion. The input looks as if you are seeing it while it is moving. A Gaussian convolution kernel is used with a specified angle to produce the effect. An instance of the `MotionBlur` class represents a `MotionBlur` effect. The effect can be configured using the three properties of the class:

- `radius`
- `angle`
- `input`

The `radius` and `input` properties work the same as respective properties for the `GaussianBlur` class, as described in the previous section. The `angle` property specifies the angle of the motion in degrees. By default, the angle is zero.

The `MotionBlur` class contains two constructors:

- `MotionBlur()`
- `MotionBlur(double angle, double radius)`

The no-args constructor creates a MotionBlur object with a default radius of 10.0px and an angle of 0.0 degrees. The other constructor lets you specify the initial value for the angle and radius, as shown in the following code:

```
// Create a MotionBlur with a 0.0 degrees angle and a 10.0 pixels
radius
MotionBlur mbl = new MotionBlur();

// Create a MotionBlur with a 30.0 degrees angle and a 20.0
pixels radius
MotionBlur mb1 = new MotionBlur(30.0, 20.0);
```

The program in Listing 20-5 shows how to use the MotionBlur effect on a Text node, with the results shown in Figure 20-13. The two sliders let you change the radius and angle properties.

### ***Listing 20-5.*** Using the MotionBlur Effect on a Text Node

```
// MotionBlurTest.java
package com.jdojo.effect;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.Slider;
import javafx.scene.effect.MotionBlur;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.HBox;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class MotionBlurTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Text t1 = new Text("Motion Blur");
        t1.setFont(Font.font(null, FontWeight.BOLD, 36));
        MotionBlur mbEffect = new MotionBlur();
        t1.setEffect(mbEffect);

        Slider radiusSlider = new Slider(0.0, 63.0, 10.0);
        radiusSlider.setMajorTickUnit(10);
        radiusSlider.setShowTickLabels(true);
        mbEffect.radiusProperty().bind(radiusSlider.valueProperty());
        mbEffect.angleProperty().bind(angleSlider.valueProperty());
    }
}
```

```

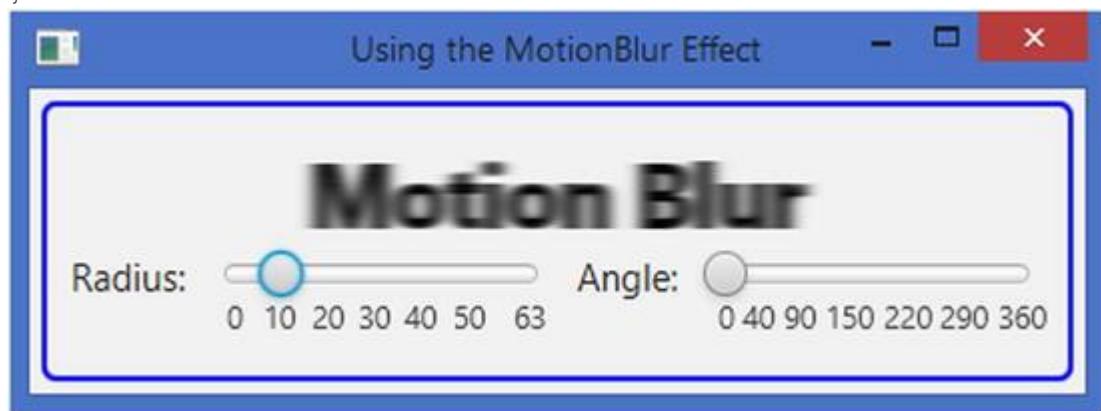
        angleSlider.setShowTickLabels(true);
        mbEffect.angleProperty().bind(angleSlider.valueProperty());
    }

    HBox pane = new HBox(10, new Label("Radius:") ,
radiusSlider,
                           new Label("Angle:") ,
angleSlider);

    BorderPane root = new BorderPane();
    root.setCenter(t1);
    root.setBottom(pane);
    root.setStyle("-fx-padding: 10;" +
                  "-fx-border-style: solid inside;" +
                  "-fx-border-width: 2;" +
                  "-fx-border-insets: 5;" +
                  "-fx-border-radius: 5;" +
                  "-fx-border-color: blue;");

    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("Using the MotionBlur Effect");
    stage.show();
}
}

```



**Figure 20-13.** Text nodes with GaussianBlur effects of varying sizes

## The Bloom Effect

The Bloom effect adds a glow to the pixels of its input that have a luminosity greater than or equal to a specified limit. Note that not all pixels in a Bloom effect are made to glow.

An instance of the Bloom class represents a Bloom effect. It contains two properties:

- threshold
- input

The `threshold` property is a number between 0.0 and 1.0. Its default value is 0.30. All pixels in the input having a luminosity greater than or equal to the `threshold` property are made to glow. The brightness of a pixel is determined by its luminosity. A pixel with a luminosity of 0.0 is not bright at all. A pixel with a luminosity of 1.0 is 100% bright. By default, all pixels having a luminosity greater than or equal to 0.3 are made to glow. A threshold of 0.0 makes all of the pixels glow. A threshold of 1.0 makes almost no pixels glow.

The `Bloom` class contains two constructors:

- `Bloom()`
- `Bloom(double threshold)`

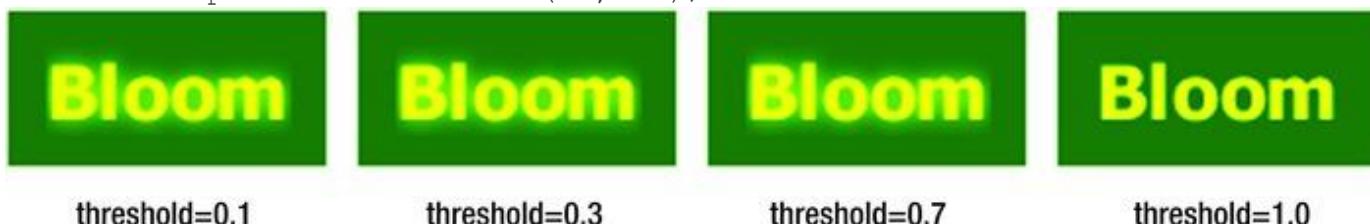
The no-args constructor creates a `Bloom` object with a default threshold of 0.30. The other constructor lets you specify the `threshold` value, as shown in the following code:

```
// Create a Bloom with threshold 0.30
Bloom b1 = new Bloom();

// Create a Bloom with threshold 0.10 - more pixels will glow.
Bloom b2 = new Bloom(0.10);
```

**Figure 20-14** shows four `Text` nodes with `Bloom` effects that have different threshold values. A `Textnode` is laid over a rectangle using a `StackPane`. Notice that the lower the threshold value, the higher the blooming effect. The following snippet of code created the first `Text` node and `Rectangle` pair from the left in Figure 20-14:

```
Text t1 = new Text("Bloom");
t1.setFill(Color.YELLOW);
t1.setFont(Font.font(null, FontWeight.BOLD, 24));
t1.setEffect(new Bloom(0.10));
Rectangle r1 = new Rectangle(100, 50, Color.GREEN);
StackPane sp1 = new StackPane(r1, t1);
```



**Figure 20-14.** Text nodes with `Bloom` effects

## The **Glow** Effect

The `Glow` effect makes the bright pixels of the input brighter. An instance of the `Glow` class represents a `Glow` effect. It contains two properties:

- level
- input

The `level` property specifies the intensity of the `Glow` effect. It is a number between 0.0 and 1.0, and its default value is 0.30. A level of 0.0 adds no glow and a level of 1.0 adds the maximum glow.

The `Glow` class contains two constructors:

- `Glow()`
- `Glow(double level)`

The no-args constructor creates a `Glow` object with a default level of 0.30. The other constructor lets you specify the level value, as shown in the following code:

```
// Create a Glow with level 0.30
Glow g1 = new Glow();

// Create a Glow with level 0.90 - more glow.
Glow g2 = new Glow(0.90);
```

Figure 20-15 shows four `Text` nodes with `Glow` effects with different level values. A `Text` node is laid over a rectangle using a `StackPane`. Notice that the higher the level value, the higher the glowing effect. The following snippet of code created the first `Text` node and `Rectangle` pair from the left in Figure 20-15:

```
Text t1 = new Text("Glow");
t1.setFill(Color.YELLOW);
t1.setFont(Font.font(null, FontWeight.BOLD, 24));
t1.setEffect(new Glow(0.10));
Rectangle r1 = new Rectangle(100, 50, Color.GREEN);
StackPane sp1 = new StackPane(r1, t1);
```



*Figure 20-15. Text nodes with `Glow` effects*

## The **Reflection** Effect

The `Reflection` effect adds a reflection of the input below the input. An instance of the `Reflection` class represents a reflection effect. The position, size, and opacity of the reflection are controlled by various properties:

- topOffset
- fraction
- topOpacity
- bottomOpacity
- input

The `topOffset` specifies the distance in pixels between the bottom of the input and the top of the reflection. By default, it is 0.0.

The `fraction` property specifies the fraction of the input height that is visible in the reflection. It is measured from the bottom. Its value can be between 0.0 and 1.0. A value of 0.0 means no reflection. A value of 1.0 means the entire input is visible in the reflection. A value of 0.25 means 25% of the input from the bottom is visible in the reflection. The default value is 0.75. The `topOpacity` and `bottomOpacity` properties specify the opacity of the reflection at its top and bottom extremes. Their values can be between 0.0 and 1.0. The default value is 0.50 for the `topOpacity` and 0.0 for the `bottomOpacity`.

The `Reflection` class contains two constructors:

- `Reflection()`
- `Reflection(double topOffset, double fraction, double topOpacity, double bottomOpacity)`

The no-args constructor creates a `Reflection` object with the default initial values for its properties. The other constructor lets you specify the initial values for the properties, as shown in the following code:

```
// Create a Reflection with default values
Reflection g1 = new Reflection();

// Create a Reflection with topOffset=2.0, fraction=0.90,
// topOpacity=1.0, and bottomOpacity=1.0
Reflection g2 = new Reflection(2.0, 0.90, 1.0, 1.0);
```

Figure 20-16 shows four `Text` nodes with `Reflection` effects configured differently. The following snippet of code creates the second `Text` node from the left, which shows the full input as the reflection:

```
Text t2 = new Text("Chatar");
t2.setFont(Font.font(null, FontWeight.BOLD, 24));
t2.setEffect(new Reflection(0.0, 1.0, 1.0, 1.0));
```



**Figure 20-16.** Text nodes with Reflection effects

## The **SepiaTone** Effect

Sepia is a reddish-brown color. Sepia toning is performed on black-and-white photographic prints to give them a warmer tone. An instance of the `SepiaTone` class represents a SepiaTone effect. It contains two properties:

- `level`
- `input`

The `level` property specifies the intensity of the `SepiaTone` effect. It is a number between 0.0 and 1.0. Its default value is 1.0. A `level` of 0.0 adds no sepia toning and a `level` of 1.0 adds the maximum sepia toning.

The `SepiaTone` class contains two constructors:

- `SepiaTone ()`
- `SepiaTone (double level)`

The no-args constructor creates a `SepiaTone` object with a default `level` of 1.0. The other constructor lets you specify the `level` value, as shown in the following code:

```
// Create a SepiaTone with level 1.0
SepiaTone g1 = new SepiaTone ();

// Create a SepiaTone with level 0.50
SepiaTone g2 = new SepiaTone(0.50);
```

The following snippet of code creates two `Text` nodes with the results shown in Figure 20-17. Notice that the higher the `level` value, the higher the sepia toning effect:

```
Text t1 = new Text("SepiaTone");
t1.setFill(Color.WHITE);
t1.setFont(Font.font(null, FontWeight.BOLD, 24));
t1.setEffect(new SepiaTone(0.50));
```

```

Rectangle r1 = new Rectangle(150, 50, Color.BLACK);
r1.setOpacity(0.50);
StackPane sp1 = new StackPane(r1, t1);

Text t2 = new Text("SepiaTone");
t2.setFill(Color.WHITE);
t2.setFont(Font.font(null, FontWeight.BOLD, 24));
t2.setEffect(new SepiaTone(1.0));
Rectangle r2 = new Rectangle(150, 50, Color.BLACK);
r2.setOpacity(0.50);
StackPane sp2 = new StackPane(r2, t2);

```



**Figure 20-17.** Text nodes with *SepiaTone* effect

## The *DisplacementMap* Effect

The *DisplacementMap* effect shifts each pixel in the input to produce an output. The name has two parts: “Displacement” and “Map.” The first part implies that the effect displaces the pixels in the input. The second part implies that the displacement is based on a map that provides a displacement factor for each pixel in the output.

An instance of the *DisplacementMap* class represents a *DisplacementMap*. The class contains several properties to configure the effect:

- *mapData*
- *scaleX*
- *scaleY*
- *offsetX*
- *offsetY*
- *wrap*
- *input*

The *mapData* property is an instance of the *FloatMap* class. A *FloatMap* is a data structure that stores up to four values for each point in a rectangular area represented by its *width* and *height* properties. For example, you can use a *FloatMap* to store four components of the color (red, green, blue, and alpha) for each pixel in a two-dimensional rectangle. Each of the four values associated with a pair of numbers in the *FloatMap* are said to be

in a band numbered 0, 1, 2, and 3. The actual meaning of the values in each band is context dependent. The following code provides an example of setting the `FloatMap` width and height:

```
// Create a FloatMap (width = 100, height = 50)
FloatMap map = new FloatMap(100, 50);
```

Now you need to populate the `FloatMap` with band values for each pair of numbers. You can use one of the following methods of the `FloatMap` class to populate it with the data:

- `setSample(int x, int y, int band, float value)`
- `setSamples(int x, int y, float s0)`
- `setSamples(int x, int y, float s0, float s1)`
- `setSamples(int x, int y, float s0, float s1, float s2)`
- `setSamples(int x, int y, float s0, float s1, float s2, float s3)`

The `setSample()` method sets the specified `value` in the specified band for the specified (x, y) location. The `setSamples()` methods sets the specified values in the bands determined by the positions of the values in the method call. That is, the first value is set for band 0, the second value for band 1, and so forth:

```
// Set 0.50f for band 0 and band 1 for each point in the map
for (int i = 0; i < 100; i++) {
    for (int j = 0; j < 50; j++) {
        map.setSamples(i, j, 0.50f, 0.50f);
    }
}
```

The `DisplacementMap` class requires that you set the `mapData` property to a `FloatMap` that contains values for band 0 and band 1 for each pixel in the output.

The `scaleX`, `scaleY`, `offsetX`, and `offsetY` are double properties. They are used in the equation (described shortly) to compute the displacement of the pixels. The `scaleX` and `scaleY` properties have 1.0 as their default values. The `offsetX` and `offsetY` properties have 0.0 as their default values.

The following equation is used to compute the pixel at (x, y) coordinates in the output. The abbreviations `dst` and `src` in the equation represent the destination and source, respectively:

```
dst[x, y] = src[x + (offsetX + scaleX * mapData[x, y][0]) * srcWidth,
                 y + (offsetY + scaleY * mapData[x, y][1]) * srcHeight]
```

If the above equation looks very complex, don't be intimidated. In fact, the equation is very simple once you read the explanation that follows. The `mapData[x, y][0]` and `mapData[x, y][1]` parts in the equation refer to the values at band 0 and band 1, respectively, in the `FloatMap` for the location at  $(x, y)$ .

Suppose you want to get the pixel for the  $(x, y)$  coordinates in the output, that is, you want to know which pixel from the input will be moved to  $(x, y)$  in the output. First, make sure you get the starting point right. To repeat, the equation starts with a point  $(x, y)$  in the output and finds the pixel at  $(x_1, y_1)$  in the input that will move to  $(x, y)$  in the output.

**Tip** Many will get the equation wrong by thinking that you start with a pixel in the input and then find its location in the output. This is not true. The equation works the other way around. It picks a point  $(x, y)$  in the output and then finds which pixel in the input will move to this point.

Below are the steps to fully explain the equation:

- You want to find the pixel in the input that will be moved to the point  $(x, y)$  in the output.
- Get the values (band 0 and band 1) from the `mapData` for  $(x, y)$ .
- Multiply the `mapData` values by the scale (`scaleX` for x coordinate and `scaleY` for y coordinate).
- Add the corresponding offset values to the values computed in the previous step.
- Multiply the previous step values with the corresponding dimensions of the input. This gives you the offset values along the x and y coordinate axes from the output  $(x, y)$  from where the pixels in the input will be moving to the  $(x, y)$  in the output.
- Add the values in the previous step to the x and y coordinates of the point in the output. Suppose these values are  $(x_1, y_1)$ . The pixel at  $(x_1, y_1)$  in the input moves to the point  $(x, y)$  in the output.

If you still have problem understanding the pixel-shifting logic, you can break the above equation into two parts:

```
x1 = x + (offsetX + scaleX * mapData[x, y][0]) * srcWidth
y1 = y + (offsetY + scaleY * mapData[x, y][1]) * srcHeight
```

You can read these equations as "The pixel at  $(x, y)$  in the output is obtained by moving the pixel at  $(x_1, y_1)$  in the input to  $(x, y)$ ."

If you leave the scale and offset values to their default:

- Use a positive value in band 0 to move the input pixels to the left.
- Use a negative value in band 0 to move the input pixels to the right.
- Use a positive value in band 1 to move the input pixels up.
- Use a negative value in band 1 to move the input pixels down.

The program in Listing 20-6 creates a `Text` node and adds a `DisplacementMap` effect to the node. In the `mapData`, it sets values, so all pixels in the top half of the input are moved to the right by 1 pixel, and all pixels in the bottom half of the input are moved to the left by 1 pixel. The `Text` node will look like the one shown in Figure 20-18.

### ***Listing 20-6.*** Using the `DisplacementMap` Effect

```
// DisplacementmapTest.java
package com.jdojo.effect;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.effect.DisplacementMap;
import javafx.scene.effect.FloatMap;
import javafx.scene.layout.HBox;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class DisplacementmapTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // Create a FloatMap
        int width = 250;
        int height = 50;
        FloatMap map = new FloatMap(width, height);

        double xDisplacement = 1.0;
        for (int i = 0; i < width; i++) {
            for (int j = 0; j < height; j++) {
                double u = xDisplacement;
                if (j < height / 2) {
                    // Move the top-half pixels to the
right // (a nagative value)
                    u = -1.0 * (u * xDisplacement
/ width);
                } else {
                    // Move the bottom-half pixels to
the left.(a positive value)
                }
            }
        }
    }
}
```

```

        u = u * xDisplacement / width;
    }

        // Set values for band 0 and 1 (x and
y axes // displacements factors).
        // Always use 0.0f for y-axis
displacement factor.
        // map.setSamples(i, j, (float)u, 0.0f);
    }
}

Text t1 = new Text("Displaced Text");
t1.setFont(Font.font(36));

DisplacementMap effect1 = new DisplacementMap();
effect1.setMapData(map);
t1.setEffect(effect1);

HBox root = new HBox(t1);
root.setStyle("-fx-padding: 10;" +
            "-fx-border-style: solid inside;" +
            "-fx-border-width: 2;" +
            "-fx-border-insets: 5;" +
            "-fx-border-radius: 5;" +
            "-fx-border-color: blue;");

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("Applying the DisplacementMap
Effect");
stage.show();
}
}

```

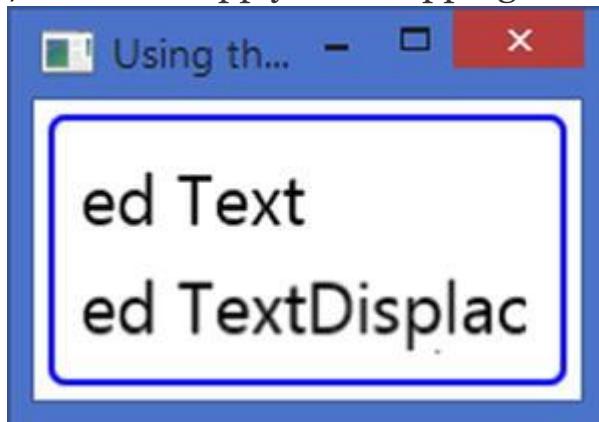


**Figure 20-18.** A Text node with DisplacementMap effect

The `DisplacementMap` class contains a `wrap` property, which is set to false by default. A pixel in the output is a pixel in the input that is moved to a new location. The location of the pixel in the input that needs to move to a new location is computed by the equation. It is possible that for some locations in the output, you do not have available pixels in the input. Suppose you have a 100px wide by 50px tall rectangle and you apply a `DisplacementMap` effect to move all pixels to the left by 50px.

The points at  $x = 75$  in the output will get the pixel at  $x = 125$  in the input. The input is only 100px wide. Therefore, for all points  $x > 50$  in the output, you will not have available pixels in the input. If the `wrap` property is set to true, when the locations of the pixels in the input to be moved are outside the input bounds, the locations are computed by taking their modulus with the corresponding dimension (width along the x axis and height for along the y axis) of the input. In the example,  $x = 125$  will be reduced to  $125 \% 100$ , which is 25 and the pixels at  $x = 25$  in the input will be moved to  $x = 75$  in the output. If the `wrap` property is false, the pixels in the output are left transparent.

Figure 20-19 shows two `Text` nodes with `DisplacementMap` effects. Pixels in both nodes are moved 100px to the left. The `Text` node at the top has the `wrap` property set to false, whereas the `Text` node at the bottom has the `wrap` property set to true. Notice that output for the bottom node is filled by wrapping the input. The program in Listing 20-7 is used to apply the wrapping effects.



**Figure 20-19.** Effects of using the `wrap` property in `DisplacementMap`

### **Listing 20-7.** Using the `wrap` Property in `DisplacementMap` Effect

```
// DisplacementMapWrap.java
package com.jdojo.effect;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.effect.DisplacementMap;
import javafx.scene.effect.FloatMap;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class DisplacementMapWrap extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }
}
```

```

    }

@Override
public void start(Stage stage) {
    // Create a FloatMap
    int width = 200;
    int height = 25;

    FloatMap map = new FloatMap(width, height);
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++) {
            // Move all pixels 100 pixels to the
left
            double u = 100.0/width;
            map.setSamples(i, j, (float)u, 0.0f);
        }
    }

    Text t1 = new Text("Displaced Text");
    t1.setFont(Font.font(24));
    DisplacementMap effect1 = new DisplacementMap();
    effect1.setMapData(map);
    t1.setEffect(effect1);

    Text t2 = new Text("Displaced Text");
    t2.setFont(Font.font(24));
    DisplacementMap effect2 = new DisplacementMap();
    effect2.setWrap(true);
    effect2.setMapData(map);
    t2.setEffect(effect2);

    VBox root = new VBox(t1, t2);
    root.setSpacing(5);
    root.setStyle("-fx-padding: 10;" +
                  "-fx-border-style: solid inside;" +
                  "-fx-border-width: 2;" +
                  "-fx-border-insets: 5;" +
                  "-fx-border-radius: 5;" +
                  "-fx-border-color: blue;");

    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("Using the warps proeprty in
DisplacementMap");
    stage.show();
}
}

```

## The **ColorInput** Effect

The **ColorInput** effect is a simple effect that fills (floods) a rectangular region with a specified paint. Typically, it is used as an input to another effect.

An instance of the `ColorInput` class represents the `ColorInput` effect. The class contains five properties that define the location, size, and the paint for the rectangular region:

- `x`
- `y`
- `width`
- `height`
- `paint`

Creating a `ColorInput` object is similar to creating a rectangle filled with the paint of the `ColorInput`. The `x` and `y` properties specify the location of the upper left corner of the rectangular region in the local coordinate system. The `width` and `height` properties specify the size of the rectangular region. The default value for `x`, `y`, `width`, and `height` is `0.0`. The `paint` property specifies the fill paint. The default value for `paint` is `Color.RED`.

You can use the following constructors to create an object of the `ColorInput` class:

- `ColorInput()`
- `ColorInput(double x, double y, double width, double height, Paint paint)`

The following snippet of code creates a `ColorInput` effect and applies it to a rectangle. The rectangle with the effect applied is shown in Figure 20-20. Note that when you apply the `ColorInput` effect to a node, all you see is the rectangular area generated by the `ColorInput` effect. As stated earlier, the `ColorInput` effect is not applied directly on nodes. Rather it is used as an input to another effect.

```
ColorInput effect = new ColorInput();
effect.setWidth(100);
effect.setHeight(50);
effect.setPaint(Color.LIGHTGRAY);

// Size of the Rectangle does not matter to the rectangular area
// of the ColorInput
Rectangle r1 = new Rectangle(100, 50);
r1.setEffect(effect);
```



**Figure 20-20.** A `ColorInput` effect applied to a rectangle

## The `ColorAdjust` Effect

The `ColorAdjust` effect adjusts the hue, saturation, brightness, and contrast of pixels by the specified delta amount. Typically, the effect is used on an `ImageView` node to adjust the color of an image.

An instance of the `ColorAdjust` class represents the `ColorAdjust` effect. The class contains five properties that define the location, size, and the paint for the rectangular region:

- `hue`
- `saturation`
- `brightness`
- `contrast`
- `input`

The `hue`, `saturation`, `brightness`, and `contrast` properties specify the delta amount by which these components are adjusted for all pixels. They range from -1.0 to 1.0. Their default values are 0.0.

The program in Listing 20-8 shows how to use the `ColorAdjust` effect on an image. It displays an image and four sliders to change the properties of the `ColorAdjust` effect. Adjust their values using the sliders to see the effects. If the program does not find the image, it prints a message and displays a `Text` node overlaying a rectangle in a `StackPane` and the effect is applied to the `StackPane`.

### **Listing 20-8.** Using the `ColorAdjust` Effect to Adjust the Color of Pixels in an Image

```
// ColorAdjustTest.java
package com.jdojo.effect;

import java.net.URL;
import javafx.application.Application;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.Slider;
import javafx.scene.effect.ColorAdjust;
import javafx.scene.image.ImageView;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Text;
```

```

import javafx.stage.Stage;

public class ColorAdjustTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        ColorAdjust effect = new ColorAdjust();

        Node node = getImageNode();
        node.setEffect(effect);

        GridPane controller = getController(effect);

        BorderPane root = new BorderPane();
        root.setCenter(node);
        root.setBottom(controller);
        root.setStyle("-fx-padding: 10;" +
                     "-fx-border-style: solid inside;" +
                     "-fx-border-width: 2;" +
                     "-fx-border-insets: 5;" +
                     "-fx-border-radius: 5;" +
                     "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Applying the ColorAdjust Effect");
        stage.show();
    }

    private Node getImageNode() {
        Node node = null;
        String path
= "\\\resources\\\\picture\\\\randomness.jpg";
        URL url
= getClass().getClassLoader().getResource(path);

        if (url != null) {
            node = new ImageView(url.toExternalForm());
        } else {
            System.out.println("Missing image file "
+ path);
            node = new StackPane(new Rectangle(100, 50,
Color.LIGHTGRAY),
                               new Text("Color Adjust"));
        }
        return node;
    }

    private GridPane getController(ColorAdjust effect) {
        Slider hueSlider = new Slider(-1.0, 1.0, 0.0);
        effect.hueProperty().bind(hueSlider.valueProperty())
;
    }
}

```

```

        Slider saturationSlider = new Slider(-1.0, 1.0,
0.0);
        effect.saturationProperty().bind(saturationSlider.va
lueProperty());

        Slider brightnessSlider = new Slider(-1.0, 1.0,
0.0);
        effect.brightnessProperty().bind(brightnessSlider.va
lueProperty());

        Slider contrastSlider = new Slider(-1.0, 1.0, 0.0);
        effect.contrastProperty().bind(contrastSlider.valueP
roperty());

        Slider[] sliders = new Slider[] {hueSlider,
saturationSlider,
                                brightnessSlider,
contrastSlider};
        for (Slider s : sliders) {
            s.setPrefWidth(300);
            s.setMajorTickUnit(0.10);
            s.setShowTickMarks(true);
            s.setShowTickLabels(true);
        }

        GridPane pane = new GridPane();
        pane.setHgap(5);
        pane.setVgap(10);
        pane.addRow(0, new Label("Hue:"), hueSlider);
        pane.addRow(1, new Label("Saturation:"), saturationSlider);
        pane.addRow(2, new Label("Brightness:"), brightnessSlider);
        pane.addRow(3, new Label("Contrast:"), contrastSlider);

        return pane;
    }
}

```

## The **ImageInput** Effect

The `ImageInput` effect works like the `ColorInput` effect. It passes the given image as an input to another effect. The given image is not modified by this effect. Typically, it is used as an input to another effect, not as an effect directly applied to a node.

An instance of the `ImageInput` class represents the `ImageInput` effect. The class contains three properties that define the location and the source of the image:

- `x`

- `y`
- `source`

The `x` and `y` properties specify the location of the upper left corner of the image in the local coordinate system of the content node on which the effect is finally applied. Their default values are `0.0`.

The `source` property specifies the `Image` object to be used.

You can use the following constructors to create an object of the `ColorInput` class:

- `ImageInput()`
- `ImageInput(Image source)`
- `ImageInput(Image source, double x, double y)`

The program in Listing 20-9 shows how to use the `ImageInput` effect. It passes an `ImageInput` as an input to a `DropShadow` effect, which is applied on a rectangle, as shown in Figure 20-21.

### ***Listing 20-9.*** Using an `ImageInput` Effect as an Input to a `DropShadow` Effect

```
// ImageInputTest.java
package com.jdojo.effect;

import java.net.URL;
import javafx.application.Application;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.effect.GaussianBlur;
import javafx.scene.effect.ImageInput;
import javafx.scene.image.Image;
import javafx.scene.layout.HBox;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class ImageInputTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

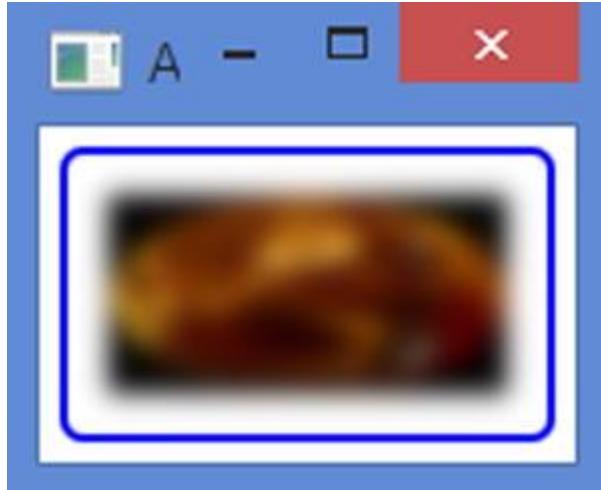
    @Override
    public void start(Stage stage) {
        String path
= "\\resources\\picture\\randomness.jpg";
        URL url
= getClass().getClassLoader().getResource(path);
```

```
Node node = null;
if (url == null) {
    node = new Text("Missing image file " + path
+ " in classpath.");
}
else {
    ImageInput imageInputEffect = new
ImageInput();
    double requestedWidth = 100;
    double requestedHeight = 50;
    boolean preserveRation = false;
    boolean smooth = true;
    Image image = new Image(url.toExternalForm(),
                           requestedWidth,
                           requestedHeight,
                           preserveRation,
                           smooth);
    imageInputEffect.setSource(image);

    node = new Rectangle(100, 50);
    GaussianBlur dsEffect = new GaussianBlur();
    dsEffect.setInput(imageInputEffect);
    node.setEffect(dsEffect);
}

HBox root = new HBox(node);
root.setStyle("-fx-padding: 10;" +
              "-fx-border-style: solid inside;" +
              "-fx-border-width: 2;" +
              "-fx-border-insets: 5;" +
              "-fx-border-radius: 5;" +
              "-fx-border-color: blue;");

Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("Applying the ImageInput Effect");
stage.show();
}
```



**Figure 20-21.** An *ImageInput* effect with a *DropShadow* effect applied to a rectangle

## The **Blend** Effect

Blending combines two pixels at the same location from two inputs to produce one composite pixel in the output. The `Blend` effect takes two input effects and blends the overlapping pixels of the inputs to produce an output. The blending of two inputs is controlled by a blending mode.

An instance of the `Blend` class represents the Blend effect. The class contains properties to specify the:

- `topInput`
- `bottomInput`
- `mode`
- `opacity`

The `topInput` and `bottomInput` properties specify the top and bottom effects, respectively. They are `null` by default.

The `mode` property specifies the blending mode, which is one of the constants defined in the `BlendMode` enum. The default is `BlendMode.SRC_OVER`. JavaFX provides 17 predefined blending modes. Table 20-1 lists all of the constants in the `BlendMode` enum with a brief description of each. All blending modes use the `SRC_OVER` rules to blend the alpha components. The `opacity` property specifies the opacity to be applied to the top input before the blending is applied. The `opacity` is 1.0 by default.

**Table 20-1.** The Constants in the *BlendMode* Enum with Their Descriptions

<b>BlendMode</b> Enum Constant	Description
ADD	It adds the color (red, green, and blue) and alpha values for the pixels and bottom inputs to get the new component value.
MULTIPLY	It multiplies the color components from two inputs.
DIFFERENCE	It subtracts the darker color components from any inputs from the lighter components of the other input to get the resulting color components.
RED	It replaces the red component of the bottom input with the red compo-

<b>BlendMode Enum Constant</b>	<b>Description</b>
	top input, leaving all other color components unaffected.
BLUE	It replaces the blue component of the bottom input with the blue component of the top input, leaving all other color components unaffected.
GREEN	It replaces the green component of the bottom input with the green component of the top input, leaving all other color components unaffected.
EXCLUSION	It multiplies the color components of the two inputs and doubles the value thus obtained is subtracted from the sum of the color components of the bottom input to get the resulting color component.
COLOR_BURN	It divides the inverse of the bottom input color components by the top input color components and inverts the result.
COLOR_DODGE	It divides the bottom input color components by the inverse of the top input color components.
LIGHTEN	It uses the lighter of the color components from the two inputs.
DARKEN	It uses the darker of the color components from the two inputs.
SCREEN	It inverts the color components from both inputs, multiplies them, and then adds the result.
OVERLAY	Depending on the bottom input color, it multiplies or screens the input color components.
HARD_LIGHT	Depending on the top input color, it multiplies or screens the input color components.
SOFT_LIGHT	Depending on the top input color, it darkens or lightens the input color components.
SRC_ATOP	It keeps the bottom input for the nonoverlapping area and the top input for the overlapping area.

BlendMode Enum Constant	Description
SRC_OVER	overlapping area.

SRC\_OVER      The top input is drawn over the bottom input. Therefore, the overlapping area shows the top input.

The program in Listing 20-10 creates two `ColorInput` effects of the same size. Their `x` and `y` properties are set in such a way that they overlap. These two effects are used as top and bottom inputs to the `Blend` effect. A combo box and a slider are provided to select the blending mode and the opacity of the top input. Figure 20-22 shows the window that results from running this code. Run the program and try selecting different blending modes to see the `Blend` effect in action.

### ***Listing 20-10.*** Using the Blend Effect

```
// BlendTest.java
package com.jdojo.effect;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ComboBox;
import javafx.scene.control.Label;
import javafx.scene.control.Slider;
import javafx.scene.effect.Blend;
import javafx.scene.effect.BlendMode;
import javafx.scene.effect.ColorInput;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

public class BlendTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        ColorInput topInput = new ColorInput(0, 0, 100, 50,
Color.LIGHTGREEN);
        ColorInput bottomInput = new ColorInput(50, 25, 100,
50, Color.PURPLE);

        // Create the Blend effect
        Blend effect = new Blend();
```

```

        effect.setTopInput(topInput);
        effect.setBottomInput(bottomInput);

        Rectangle rect = new Rectangle(150, 75);
        rect.setEffect(effect);

        GridPane controller = this.getController(effect);

        HBox root = new HBox(rect, controller);
        root.setSpacing(30);
        root.setStyle("-fx-padding: 10;" +
                      "-fx-border-style: solid inside;" +
                      "-fx-border-width: 2;" +
                      "-fx-border-insets: 5;" +
                      "-fx-border-radius: 5;" +
                      "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Applying the Blend Effect");
        stage.show();
    }

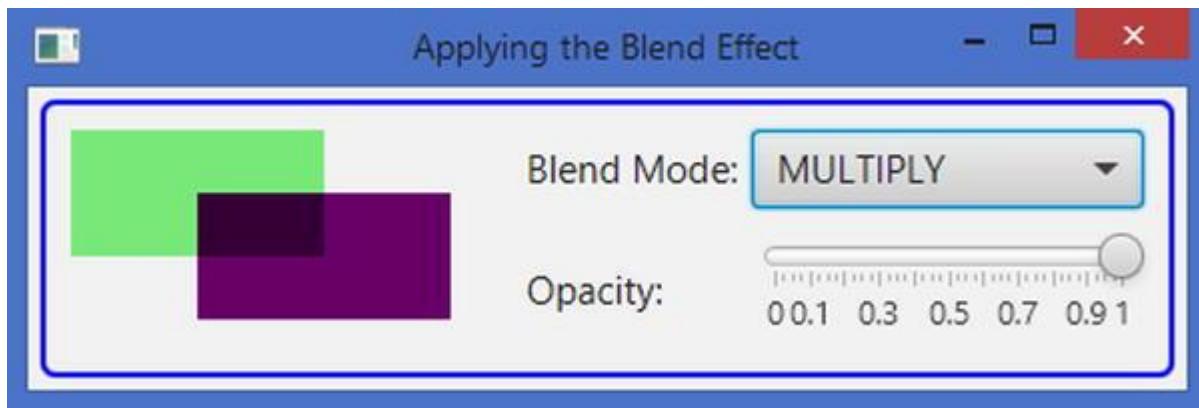
    private GridPane getController(Blend effect) {
        ComboBox<BlendMode> blendModeList = new
        ComboBox<>();
        blendModeList.setValue(effect.getMode());
        blendModeList.getItems().addAll(BlendMode.values());
        effect.modeProperty().bind(blendModeList.valuePro
        ty());

        Slider opacitySlider = new Slider (0, 1.0, 1.0);
        opacitySlider.setMajorTickUnit(0.10);
        opacitySlider.setShowTickMarks(true);
        opacitySlider.setShowTickLabels(true);
        effect.opacityProperty().bind(opacitySlider.valuePro
        perty());

        GridPane pane = new GridPane();
        pane.setHgap(5);
        pane.setVgap(10);
        pane.addRow(0, new Label("Blend Mode:"),
blendModeList);
        pane.addRow(1, new Label("Opacity:"),
opacitySlider);

        return pane;
    }
}

```



**Figure 20-22.** The Blend effect

## The **Lighting** Effect

The Lighting effect, as the name suggests, simulates a light source shining on a specified node in a scene to give the node a 3D look. A Lighting effect uses a light source, which is an instance of the Light class, to produce the effect. Different types of configurable lights are available. If you do not specify a light source, the effect uses a default light source.

An instance of the Lighting class represents a Lighting effect. The class contains two constructors:

- Lighting()
- Lighting(Light light)

The no-args constructor uses a default light source. The other constructor lets you specify a light source.

Applying a Lighting effect to a node may be a simple or complex task depending on the type of effect you want to achieve. Let's look at a simple example. The following snippet of code applies a Lightingeffect to a Text node to give it a 3D look, as shown in Figure 20-23:

```
// Create a Text Node
Text text = new Text("Chatar");
text.setFill(Color.RED);
text.setFont(Font.font(null, FontWeight.BOLD, 72));
HBox.setMargin(text, new Insets(10));

// Set a Lighting effect to the Text node
text.setEffect(new Lighting());
```

**Chatar**

**Figure 20-23.** A `Text` node with a `Lighting` effect using the default for the light source

In the above example, adding the `Lighting` effect is as simple as creating an object of the `Lighting` class and setting it as the effect for the `Text` node. I will discuss some complex `Lighting` effects later. The `Lighting` class contains several properties to configure the effect:

- `contentInput`
- `surfaceScale`
- `bumpInput`
- `diffuseConstant`
- `specularConstant`
- `specularExponent`
- `light`

If you use a chain of effects, the `contentInput` property specifies the input effect to the `Lighting` effect. This property is named as `input` in all other effects discussed earlier. I will not discuss this property further in this section. Please refer to the section “Chaining Effects” for more details on how to use this property.

### Customizing the Surface Texture

The `surfaceScale` and `bumpInput` properties are used to provide texture to a 2D surface to make it look like a 3D surface. Pixels, based on their opacity, look high or low to give the surface a texture. Transparent pixels appear low and opaque pixels appear raised.

The `surfaceScale` property lets you control the surface roughness. Its value ranges from 0.0 to 10.0. The default is 1.5. For a higher `surfaceScale`, the surface appears rougher, giving it a more 3D look.

You can pass an `Effect` as an input to the `Lighting` effect using its `bumpInput` property. The opacity of the pixels in the `bumpInput` is used to obtain the height of the pixels of the lighted surface, and then the `surfaceScale` is applied to increase the roughness.

If `bumpInput` is `null`, the opacity of the pixels from the node on which the effect is applied is used to generate the roughness of the surface. By default, a `Shadow` effect with a radius of 10 is used as the `bumpInput`. You can use an `ImageInput`, a blur effect, or any other effect as the `bumpInput` for a `Lighting` effect.

The program in Listing 20-11 displays a `Text` node with a `Lighting` effect. The `bumpInput` is set to `null`. It provides a check box to set a `GaussianBlur` effect as the `bumpInput` and a slider to

adjust the `surfaceScale` value. Figure 20-24 shows two screenshots: one without a bump input and another with a bump input. Notice the difference in the surface texture.

### ***Listing 20-11.*** Using the `surfaceScale` and `bumpInput` Properties

```
// SurfaceTexture.java
package com.jdojo.effect;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.CheckBox;
import javafx.scene.control.Slider;
import javafx.scene.effect.GaussianBlur;
import javafx.scene.effect.Lighting;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.scene.text.TextBoundsType;

import javafx.stage.Stage;

public class SurfaceTexture extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Text text = new Text();
        text.setText("Texture");
        text.setFill(Color.RED);
        text.setFont(Font.font(null, FontWeight.BOLD, 72));
        text.setBoundsType(TextBoundsType.VISUAL);

        Lighting effect = new Lighting();
        effect.setBumpInput(null); // Remove the default
        bumpInput
        text.setEffect(effect);

        // Let the user choose to use a bumpInput
        CheckBox bumpCbx = new CheckBox("Use a GaussianBlur
        Bump Input?");
        bumpCbx.selectedProperty().addListener((prop,
        oldValue, newValue) -> {
            if (newValue) {
                effect.setBumpInput(new
                GaussianBlur(20));
            } else {
                effect.setBumpInput(null);
            }
        });
    }
}
```

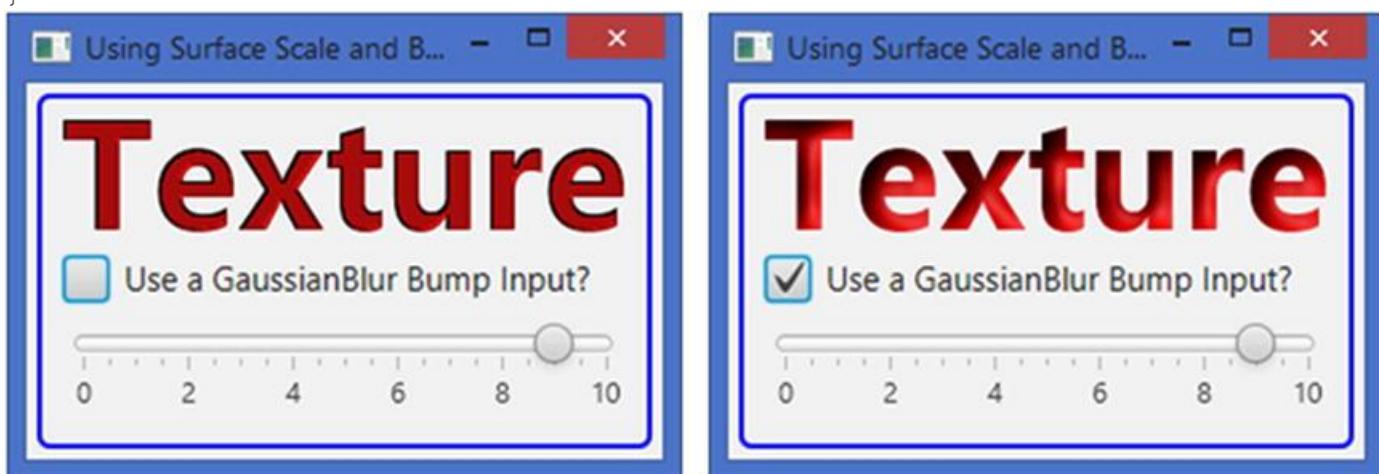
```

        });

        // Let the user select a surfaceScale
        Slider scaleSlider = new Slider(0.0, 10.0, 1.5);
        effect.surfaceScaleProperty().bind(scaleSlider.value
Property());
        scaleSlider.setShowTickLabels(true);
        scaleSlider.setMajorTickUnit(2.0);
        scaleSlider.setShowTickMarks(true);

        VBox root = new VBox(10, text, bumpCbx,
scaleSlider);
        root.setStyle("-fx-padding: 10;" +
                  "-fx-border-style: solid inside;" +
                  "-fx-border-width: 2;" +
                  "-fx-border-insets: 5;" +
                  "-fx-border-radius: 5;" +
                  "-fx-border-color: blue;");
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Using Surface Scale and Bump
Input");
        stage.show();
    }
}

```



**Figure 20-24.** The effects of `surfaceScale` and `bumpInput` on a Lighting effect on a Text node

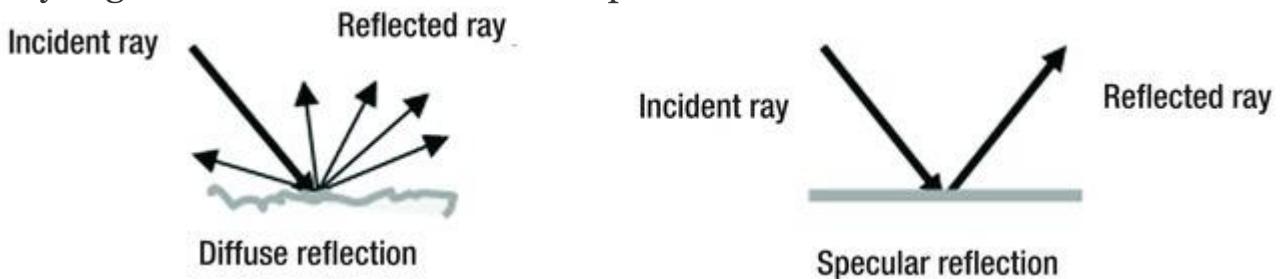
### Understanding Reflection Types

When light falls on an opaque surface, part of light is absorbed, part is transmitted, and some is reflected. A 3D look is achieved by showing part of the surface brighter and part shadowy. You see the reflected light from the surface. The 3D look varies depending on the light source and the way the node surface reflects the light. The structure of the surface at the microscopic level defines the details of the reflection, such as the

intensity and directions. Among several reflection types, two types are worth mentioning at this point: diffuse reflection and specular reflection.

In a *diffuse* reflection, the surface reflects an incident ray of light at many angles. That is, a diffuse reflection scatters a ray of light by reflecting it in all directions. A perfect diffuse reflection reflects light equally in all directions. The surface using a diffuse reflection appears to be equally bright from all directions. This does not mean that the entire diffuse surface is visible. The visibility of an area on a diffuse surface depends on the direction of the light and the orientation of the surface. The brightness of the surface depends on the surface type itself and the intensity of the light. Typically, a rough surface, for example, clothing, paper, or plastered walls, reflects light using a diffuse reflection. Surfaces may appear smooth to the eyes, for example, paper or clothing, but they are rough at the microscopic level, and they reflect light diffusively.

In a *specular* reflection, the surface reflects a ray of light in exactly one direction. That is, there is a single reflected ray for one incident ray. A smooth surface at the microscopic level, for example, mirrors or polished marbles, produces a specular reflection. Some smooth surfaces may not be 100% smooth at the microscopic level, and they may reflect part of the light diffusively as well. Specular reflection produces a brighter surface compared to diffuse reflection. Figure 20-25 depicts the ways light is reflected in diffuse and specular reflections.



**Figure 20-25.** Diffuse and specular reflection type

Three properties of the `Lighting` class are used to control the size and intensity of the reflection:

- `diffuseConstant`
- `specularConstant`
- `specularExponent`

The properties are of the double type. The `diffuseConstant` is used for diffuse reflection.

The `specularConstant` and `specularExponent` are used for specular reflection. The `diffuseConstant` property specifies a multiplier for the diffuse reflection intensity. Its value ranges from 0.0 to

2.0 with a default of 1.0. A higher value makes the surface brighter. The `specularConstant` property specifies the fraction of the light to which the specular reflection applies. Its value ranges from 0.0 to 2.0 with a default value of 0.30. A higher value means a bigger-sized specular highlight. The `specularExponent` specifies the shininess of the surface. A higher value means a more intense reflection and the surface looks shinier. The `specularExponent` ranges from 0.0 to 40.0 with a default value of 20.0.

**Listing 20-12** contains the code for a utility class that binds the properties of the `Lighting` class to some controls that will be used to control the properties in the examples discussed later.

### **Listing 20-12.** A Utility Class that Creates a Set of Controls Bound to the Properties of a Lighting Instance

```
// LightingUtil.java
package com.jdojo.effect;

import javafx.beans.property.DoubleProperty;
import javafx.scene.control.Label;
import javafx.scene.control.Slider;
import javafx.scene.effect.Lighting;
import javafx.scene.layout.GridPane;

public class LightingUtil {
    public static GridPane getPropertyControllers(Lighting
effect) {
        Slider surfaceScaleSlider = getSlider(0.0, 10.0,
                effect.getSurfaceScale(),
effect.surfaceScaleProperty());
        Slider diffuseConstantSlider = getSlider(0.0, 2.0,
                effect.getDiffuseConstant(),
effect.diffuseConstantProperty());
        Slider specularConstantSlider = getSlider(0.0, 2.0,
                effect.getSpecularConstant(),
effect.specularConstantProperty());
        Slider specularExponentSlider = getSlider(0.0, 40.0,
                effect.getSpecularExponent(),
effect.specularExponentProperty());

        GridPane pane = new GridPane();
        pane.setHgap(5);
        pane.setVgap(5);
        pane.addRow(0, new Label("Surface Scale:"),  
surfaceScaleSlider);
        pane.addRow(1, new Label("Diffuse Constant:"),  
diffuseConstantSlider);
        pane.addRow(2, new Label("Specular Constant:"),  
specularConstantSlider);
        pane.addRow(3, new Label("Specular Exponent:"),
```

```

        specularExponentSlider);

        return pane;
    }

    public static Slider getSlider(double min, double max,
double value,
        DoubleProperty prop) {
        Slider slider = new Slider(min, max, value);
        slider.setShowTickMarks(true);
        slider.setShowTickLabels(true);
        slider.setMajorTickUnit(max / 4.0);
        prop.bind(slider.valueProperty());
        return slider;
    }
}

```

The program in Listing 20-13 uses the utility class to bind the properties of a Lighting effect to UI controls. It displays a window as shown in Figure 20-26. Change the reflection properties using the sliders to see their effects.

### ***Listing 20-13.*** Controlling Reflection's Details

```

// ReflectionTypeTest.java
package com.jdojo.effect;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.effect.Lighting;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.scene.text.TextBoundsType;
import javafx.stage.Stage;

public class ReflectionTypeTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Text text = new Text();
        text.setText("Chatar");
        text.setFill(Color.RED);
        text.setFont(Font.font("null", FontWeight.BOLD,
72));
        text.setBoundsType(TextBoundsType.VISUAL);
    }
}

```

```

        Rectangle rect = new Rectangle(300, 100);
        rect.setFill(Color.LIGHTGRAY);

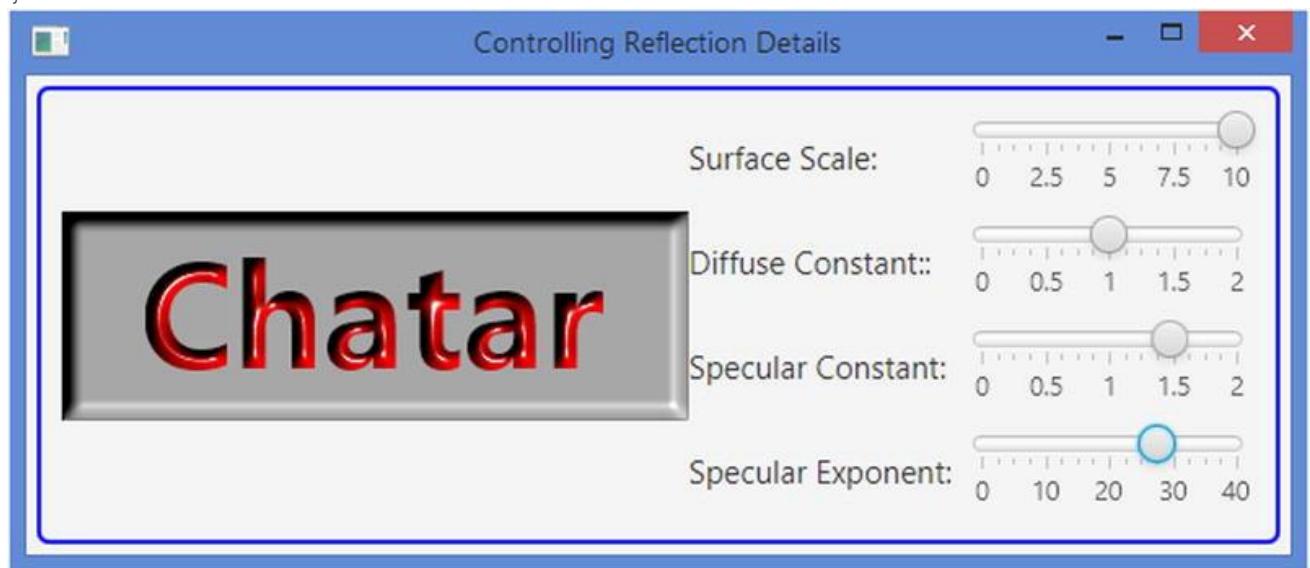
        // Set the same Lighting effect to both Rectangle
        and Text nodes
        Lighting effect = new Lighting();
        text.setEffect(effect);
        rect.setEffect(effect);

        StackPane sp = new StackPane(rect, text);

        GridPane controlsrPane
= LightingUtil.getPropertyControllers(effect);
        BorderPane root = new BorderPane();
        root.setCenter(sp);
        root.setRight(controlsrPane);
        root.setStyle("-fx-padding: 10;" +
                  "-fx-border-style: solid inside;" +
                  "-fx-border-width: 2;" +
                  "-fx-border-insets: 5;" +
                  "-fx-border-radius: 5;" +
                  "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Controlling Reflection Details");
        stage.show();
    }
}

```

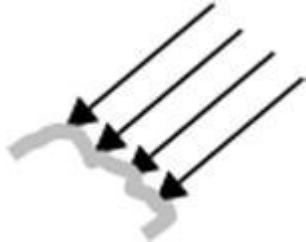


**Figure 20-26.** Effects of reflection properties on lighting nodes

### Understanding the Light Source

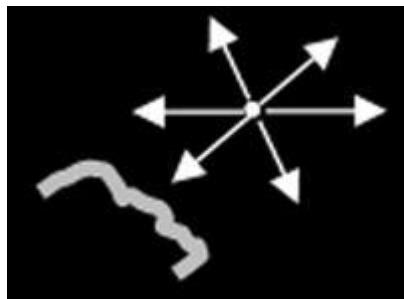
JavaFX provides three built-in light sources: distant light, point light, and spot light. A *distant* light is also known as

a *directional* or *linear* light. A distant light source emanates parallel rays of light in a *specific direction* on the entire surface uniformly. The sun is a perfect example of a distant light source for the lighted surface of an object on the earth. The light source is so distant from the lighted object that the rays are almost parallel. A distant light source lights a surface uniformly, irrespective of its distance from the surface. This does not mean that the entire object is lighted. For example, when you stand in sunlight, not all parts of your body are lighted. However, the lighted part of your body has uniform light. The lighted part of an object depends on the direction of the light. Figure 20-27 shows a distant light hitting some part of the surface of an object. Notice that the rays of light are seen, not the light source itself, because, for a distant light, only the direction of the light is important, not the distance of the light source from the lighted object.



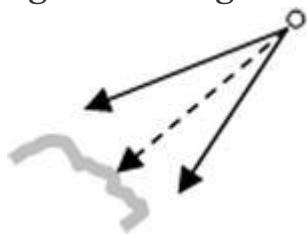
**Figure 20-27.** A distant light hitting the surface of an object

A *point* light source emanates rays of light in all directions from an infinitesimally small point in a 3D space. Theoretically, the light source has no dimension. It emanates light uniformly in all directions. Therefore, unlike the distant light, the direction of the point light source relative to the lighted object is immaterial. Bare light bulbs, stars (excluding the sun, which serves like a distant light), and candlelight are examples of point light sources. The intensity of a point light hitting a surface decreases with the square of the distance between the surface and the point light source. If a point light is very close to the surface, it creates a hotspot, which is a very bright point on the surface. To avoid hotspots, you need to move the light source a little away from the surface. A point light source is defined at a specific point in a 3D space, for example, using x, y, and z coordinates of the point. Figure 20-28 shows a point light radiating rays in all directions. The point on the object surface closest to the light will be illuminated the most.



**Figure 20-28.** A point light hitting the surface of an object

A spot light is a special type of a point light. Like a point light, it emanates rays of light radially from an infinitesimally small point in a 3D space. Unlike a point light, the radiation of light rays is confined to an area defined by a cone—the light source being at the vertex of the cone emanating light toward its base, as shown in Figure 20-29. Examples of spot lights are car headlights, flashlights, spotlights, and desk lights with lampshades. A spot light is aimed at a point on the surface, which is the point on the surface where the cone axis is located. The cone axis is the line joining the vertex of the cone to the center of the base of the cone. In Figure 20-29, the cone axis is shown with a dashed arrow. The effect of a spot light is defined by the position of the vertex of the cone, the cone angle, and the rotation of the cone. The rotation of the cone determines the point on the surface that is intersected by the cone axis. The angle of the cone controls the area of the lighted area. The intensity of a spot light is highest along the cone axis. You can simulate a distant light using a spot light if you pull the spot light “far” back, so the rays of light reaching the surface are parallel.



**Figure 20-29.** A spot light hitting the surface of an object

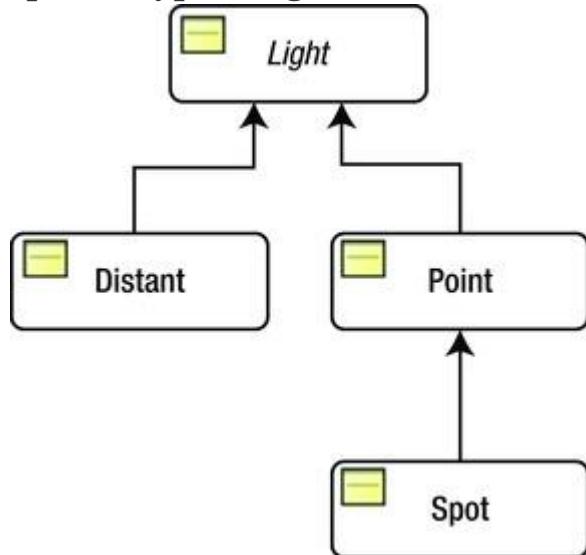
A light source is an instance of the abstract `Light` class. A light has a color, which is specified by using the `color` property of the `Light` class. For example, using a red color `Light` will make a `Text` node with a white fill look red.

There are three subclasses of the `Light` class to represent specific types of light source. The subclasses are static inner classes of the `Light` class:

- `Light.Distant`

- Light.Point
- Light.Spot

A class diagram for classes representing light sources is shown in Figure 20-30. The Light.Spot class inherits from the Light.Point class. Classes define properties to configure the specific type of light sources.



**Figure 20-30.** A class diagram for classes representing a light source

**Tip** When you do not provide a light source for a lighting effect, a distant light is used, which is an instance of the Light.Distant class.

## Using a Distant Light Source

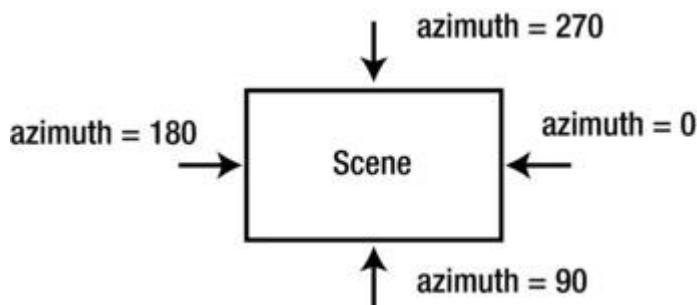
An instance of the Light.Distant class represents a distant light source. The class contains two properties to specify the direction of the light source:

- azimuth
- elevation

Both properties are of the double type. Their values are specified in degrees. Both properties are used together to position the light source in a 3D space in a specific direction. By default, their values are 45 degrees. They do not have maximum and minimum values. Their values are computed using modulo 360. For example, an azimuth value of 400 is effectively 40 (400 modulo 360 = 40).

The azimuth property specifies the direction angle in the XY plane. A positive value is measured clockwise and a negative value is measured counterclockwise. A 0 value for the azimuth is located at the 3 o'clock

position, 90 at 6 o'clock, 180 at 9 o'clock, 270 at 12 o'clock, and 360 at 3 o'clock. An azimuth of -90 will be located at 12 o'clock. Figure 20-31 shows the location of the distant light in the XY plane for different azimuth values.



**Figure 20-31.** Determining the direction of the distant light in the XY plane using the azimuth value

The elevation property specifies the direction angle of the light source in the YZ plane. The elevation property values of 0 and 180 make the light source stay on the XY plane. An elevation of 90 puts the light source in front of the scene and the entire scene is lighted. An elevation greater than 180 and less than 360 puts the light source behind the scene making it appear dark (without light).

The Light.Distant class contains two constructors:

- Light.Distant()
- Light.Distant(double azimuth, double elevation, Color color)

The no-args constructor uses 45.0 degrees for azimuth and elevation and Color.WHITE as the light color. The other constructor lets you specify these properties.

The program in Listing 20-14 shows how to use a Light.Distant light. It displays a window that lets you set the direction for a distant light shining on a rectangle and a Text node. Figure 20-32 shows an example of a text and rectangle with a distant light.

### **Listing 20-14.** Using a Distant Light Source

```
// DistantLightTest.java
package com.jdojo.effect;

import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Label;
```

```

import javafx.scene.control.Slider;
import javafx.scene.effect.Light;
import javafx.scene.effect.Lighting;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.scene.text.TextBoundsType;
import javafx.stage.Stage;

public class DistantLightTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // Create a light source and position it in the
space
        Light.Distant light = new Light.Distant(45.0, 60.0,
Color.WHITE);

        // Create a Lighting effect with the light source
        Lighting effect = new Lighting();
        effect.setLight(light);
        effect.setSurfaceScale(8.0);

        Text text = new Text();
        text.setText("Distant");
        text.setFill(Color.RED);
        text.setFont(Font.font("null", FontWeight.BOLD,
72));
        text.setBoundsType(TextBoundsType.VISUAL);

        Rectangle rect = new Rectangle(300, 100);
        rect.setFill(Color.LIGHTGRAY);

        // Set the same Lighting effect to both Rectangle
and Text nodes
        text.setEffect(effect);
        rect.setEffect(effect);

        StackPane sp = new StackPane(rect, text);
        BorderPane.setMargin(sp, new Insets(5));
        GridPane lightDirectionController
= this.getDistantLightUI(light);
        GridPane controlsrPane
= LightingUtil.getPropertyControllers(effect);

        BorderPane root = new BorderPane();
        root.setCenter(sp);
    }
}

```

```

        root.setRight(controllsrPane);
        root.setBottom(lightDirectionController);
        root.setStyle("-fx-padding: 10;" +
                      "-fx-border-style: solid inside;" +
                      "-fx-border-width: 2;" +
                      "-fx-border-insets: 5;" +
                      "-fx-border-radius: 5;" +
                      "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Configuring a Distant Light");
        stage.show();
    }

    private GridPane getDistantLightUI(Light.Distant light) {
        Slider azimuthSlider = LightingUtil.getSlider(0.0,
360.0,
                           light.getAzimuth(),
light.azimuthProperty());
        Slider elevationSlider = LightingUtil.getSlider(0.0,
360.0,
                           light.getElevation(),
light.elevationProperty());

        GridPane pane = new GridPane();
        pane.setHgap(5);
        pane.setVgap(5);
        pane.addRow(0, new Label("Azimuth:"),
azimuthSlider);
        pane.addRow(1, new Label("Elevation:"),
elevationSlider);

        return pane;
    }
}

```



**Figure 20-32.** A distant light lighting a Text node and a rectangle

## Using a Point Light Source

An instance of the `Light.Point` class represents a point light source. The class contains three properties to specify the position of the light source in space: `x`, `y`, and `z`. The `x`, `y`, and `z` properties are the `x`, `y`, and `z` coordinates of point where the point light is located in the space. If you

set the `z` property to `0.0`, the light source will be in the plane of the scene showing as a very tiny bright point lighting a very small area. As the `z` value increases, the light source moves away from the scene plane, lighting more area on the scene. A negative value of `z` will move the light source behind the scene, leaving it with no light, and the scene will look completely dark.

The `Light.Point` class contains two constructors:

- `Light.Point()`
- `Light.Point(double x, double y, double z, Color color)`

The no-args constructor places the point light at `(0, 0, 0)` and uses a `Color.WHITE` color for the light. The other constructor lets you specify the location and the color of the light source.

The program in Listing 20-15 shows how to use a `Light.Point` light. It displays a window with sliders at the bottom to change the location of the point light source. As the point light source moves away from the scene, some area on the scene will be brighter than the other area. Figure 20-33 shows an example of a `Text` node overlaid on a rectangle being lighted by a point light.

### ***Listing 20-15.*** Using a Point Light Source

```
// PointLightTest.java
package com.jdojo.effect;

import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.Slider;
import javafx.scene.effect.Light;
import javafx.scene.effect.Lighting;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.scene.text.TextBoundsType;
import javafx.stage.Stage;

public class PointLightTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }
}
```

```

    }

@Override
public void start(Stage stage) {
    // Create a light source and position it in the
space
    Light.Point light = new Light.Point(150.0, 50.0,
50.0, Color.WHITE);

    // Create a Lighting effect with the light source
    Lighting effect = new Lighting();
    effect.setLight(light);
    effect.setSurfaceScale(8.0);

    Text text = new Text();
    text.setText("Point");
    text.setFill(Color.RED);
    text.setFont(Font.font("null", FontWeight.BOLD,
72));
    text.setBoundsType(TextBoundsType.VISUAL);

    Rectangle rect = new Rectangle(300, 100);
    rect.setFill(Color.LIGHTGRAY);

    // Set the same Lighting effect to both Rectangle
and Text nodes
    text.setEffect(effect);
    rect.setEffect(effect);

    StackPane sp = new StackPane(rect, text);
    BorderPane.setMargin(sp, new Insets(5));
    GridPane lightDirectionController
= this.getPointLightUI(light);
    GridPane controlsrPane
= LightingUtil.getPropertyControllers(effect);

    BorderPane root = new BorderPane();
    root.setCenter(sp);
    root.setRight(controlsrPane);
    root.setBottom(lightDirectionController);
    root.setStyle("-fx-padding: 10;" +
"-fx-border-style: solid inside;" +
"-fx-border-width: 2;" +
"-fx-border-insets: 5;" +
"-fx-border-radius: 5;" +
"-fx-border-color: blue;");

    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("Configuring a Point Light");
    stage.show();
}

private GridPane getPointLightUI(Light.Point light) {
    Slider xSlider = LightingUtil.getSlider(-200.0,

```

```

200.0,
        light.getX(), light.xProperty());
Slider ySlider = LightingUtil.getSlider(-200.0,
200.0,
        light.getY(), light.yProperty());
Slider zSlider = LightingUtil.getSlider(-200.0,
200.0,
        light.getZ(), light.zProperty());

GridPane pane = new GridPane();
pane.setHgap(5);
pane.setVgap(5);
pane.addRow(0, new Label("x:"), xSlider);
pane.addRow(1, new Label("y:"), ySlider);
pane.addRow(2, new Label("z:"), zSlider);

return pane;
}
}

```



**Figure 20-33.** A point light lighting a `Text` node and a rectangle

## Using a Spot Light Source

An instance of the `Light.Spot` class represents a spot light source. The class inherits from the `Light.Point` class. The inherited properties (`x`, `y`, and `z`) from the `Light.Point` class specify the location of the light source, which coincides with the vertex of the cone.

The `Light.Spot` class contains four properties to specify the position of the light source in space:

- `pointsAtX`
- `pointsAtY`
- `pointsAtZ`
- `specularExponent`

The `pointsAtX`, `pointsAtY`, and `pointsAtZ` properties specify a point in the space to set the direction of the light. A line starting from  $(x, y, z)$  and going toward  $(\text{pointsAt}X, \text{pointsAt}Y, \text{pointsAt}Z)$  is the cone axis, which is also the direction of the light. By default, they are set to `0.0`. The `specularExponent` property defines the focus of the

light (the width of the cone), which ranges from 0.0 to 4.0. The default is 1.0. The higher the value for the `specularExponent`, the narrower the cone is and the more focused light will be on the scene.

The `Light.Spot` class contains two constructors:

- `Light.Spot()`
- `Light.Spot(double x, double y, double z, double specularExponent, Color color)`

The no-args constructor places the light at (0, 0, 0) and uses a `Color.WHITE` color for the light. Because the default values for `pointsAtX`, `pointsAtY`, and `pointsAtZ` are 0.0, the light does not have a direction. The other constructor lets you specify the location and the color of the light source. The cone axis will pass from the specified (x, y, z) to (0, 0, 0).

The program in Listing 20-16 shows how to use a `Light.Spot` light. It displays a window that lets you configure the location, direction, and focus of the light using sliders at the bottom. Figure 20-34 shows an example of a `Light.Spot` light focused almost in the middle of the rectangle.

### ***Listing 20-16.*** Using a Spot Light Source

```
// SpotLightTest.java
package com.jdojo.effect;

import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.Slider;
import javafx.scene.effect.Light;
import javafx.scene.effect.Lighting;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.scene.text.TextBoundsType;
import javafx.stage.Stage;

public class SpotLightTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }
}
```

```

@Override
public void start(Stage stage) {
    // Create a light source and position it in the
space
    Light.Spot light = new Light.Spot(150.0, 50.0, 50.0,
1.0, Color.WHITE);

    // Create a Lighting effect with the light source
    Lighting effect = new Lighting();
    effect.setLight(light);
    effect.setSurfaceScale(8.0);

    Text text = new Text();
    text.setText("Spot");
    text.setFill(Color.RED);
    text.setFont(Font.font("null", FontWeight.BOLD,
72));
    text.setBoundsType(TextBoundsType.VISUAL);

    Rectangle rect = new Rectangle(300, 100);
    rect.setFill(Color.LIGHTGRAY);

    // Set the same Lighting effect to both Rectangle
and Text nodes
    text.setEffect(effect);
    rect.setEffect(effect);

    StackPane sp = new StackPane(rect, text);
    BorderPane.setMargin(sp, new Insets(5));
    GridPane lightDirectionController
= this.getPointLightUI(light);
    GridPane controllersrPane
= LightingUtil.getPropertyControllers(effect);

    BorderPane root = new BorderPane();
    root.setCenter(sp);
    root.setRight(controllersrPane);
    root.setBottom(lightDirectionController);
    root.setStyle("-fx-padding: 10;" +
"-fx-border-style: solid inside;" +
"-fx-border-width: 2;" +
"-fx-border-insets: 5;" +
"-fx-border-radius: 5;" +
"-fx-border-color: blue;");

    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("Configuring a Spot Light");
    stage.show();
}

private GridPane getPointLightUI(Light.Spot light) {
    Slider xSlider = LightingUtil.getSlider(-200.0,
200.0,

```

```

        light.getX(), light.xProperty());
Slider ySlider = LightingUtil.getSlider(-200.0,
200.0,
        light.getY(), light.yProperty());
Slider zSlider = LightingUtil.getSlider(-200.0,
200.0,
        light.getZ(), light.zProperty());

Slider pointsAtXSlider = LightingUtil.getSlider(-
200.0, 200.0,
        light.getPointsAtX(),
light.pointsAtXProperty());
Slider pointsAtYSlider = LightingUtil.getSlider(-
200.0, 200.0,
        light.getPointsAtY(),
light.pointsAtYProperty());
Slider pointsAtZSlider = LightingUtil.getSlider(-
200.0, 200.0,
        light.getPointsAtZ(),
light.pointsAtZProperty());

Slider focusSlider = LightingUtil.getSlider(0.0,
4.0,
        light.getSpecularExponent(),
light.specularExponentProperty());

GridPane pane = new GridPane();
pane.setHgap(5);
pane.setVgap(5);
pane.addRow(0, new Label("x:"), xSlider);
pane.addRow(1, new Label("y:"), ySlider);
pane.addRow(2, new Label("z:"), zSlider);
pane.addRow(3, new Label("PointsAtX:"), pointsAtXSlider);
pane.addRow(4, new Label("PointsAtY:"), pointsAtYSlider);
pane.addRow(5, new Label("PointsAtZ:"), pointsAtZSlider);
pane.addRow(6, new Label("Focus:"), focusSlider);

return pane;
}
}

```



**Figure 20-34.** A spot light lighting a Text node and a rectangle

## The *PerspectiveTransform* Effect

A *PerspectiveTransform* effect gives a 2D node a 3D look by mapping the corners to different locations. The straight lines in the original nodes remain straight. However, parallel lines in the original nodes may not necessarily remain parallel.

An instance of the *PerspectiveTransform* class represents a *PerspectiveTransform* effect. The class contains eight properties to specify the x and y coordinates of four corners:

- `ulx`
- `uly`
- `urx`
- `ury`
- `lrx`
- `lry`
- `llx`
- `lly`

The first letter in the property names (u or l) indicates upper and lower. The second letter in the property names (l or r) indicates left and right. The last letter in the property names (x or y) indicates the x or y coordinate of a corner. For example, `urx` indicates the x coordinate of the upper right corner.

**Tip** The *PerspectiveTransform* class also contains an input property to specify the input effect to it in a chain of effects.

The *PerspectiveTransform* class contains two constructors:

- `PerspectiveTransform()`
- `PerspectiveTransform(double ulx, double uly, double urx, double ury, double lrx, double lry, double llx, double lly)`

The no-args constructor creates a *PerspectiveTransform* object with all new corners at (0, 0). If you set the object as an effect to a node, the node will be reduced to a point, and you will not be able to see the node. The other constructor lets you specify the new coordinates for the four corners of the node.

The program in Listing 20-17 creates two sets of a `Text` node and a rectangle. It adds two sets to two different groups. It applies a *PerspectiveTransform* effect on the second group. Both groups are shown in Figure 20-35. The group on the left shows the original nodes; the group on the right has the effect applied to it.

### **Listing 20-17.** Using the PerspectiveTransform Effect

```
// PerspectiveTransformTest.java
package com.jdojo.effect;

import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.effect.PerspectiveTransform;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class PerspectiveTransformTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // Create the effect and set the mapping for the
corners
        PerspectiveTransform effect = new
PerspectiveTransform();
        effect.setUlx(0.0);
        effect.setUly(0.0);
        effect.setUrx(250.0);
        effect.setUry(20.0);
        effect.setLrx(310.0);
        effect.setLry(60.0);
        effect.setLlx(20.0);
        effect.setLly(60.0);

        // Create two rectangles and two Text nodes. Apply
effects
        // to one set and show another set without effect
        Rectangle rect1 = new Rectangle(200, 60,
Color.LIGHTGRAY);
        Rectangle rect2 = new Rectangle(200, 60,
Color.LIGHTGRAY);

        Text text1 = new Text();
        text1.setX(20);
        text1.setY(40);
        text1.setText("Welcome");
        text1.setFill(Color.RED);
        text1.setFont(Font.font(null, FontWeight.BOLD, 36));

        System.out.println(text1.getLayoutBounds());
    }
}
```

```

Text text2 = new Text();
text2.setX(20);
text2.setY(40);
text2.setText("Welcome");
text2.setFill(Color.RED);
text2.setFont(Font.font(null, FontWeight.BOLD, 36));

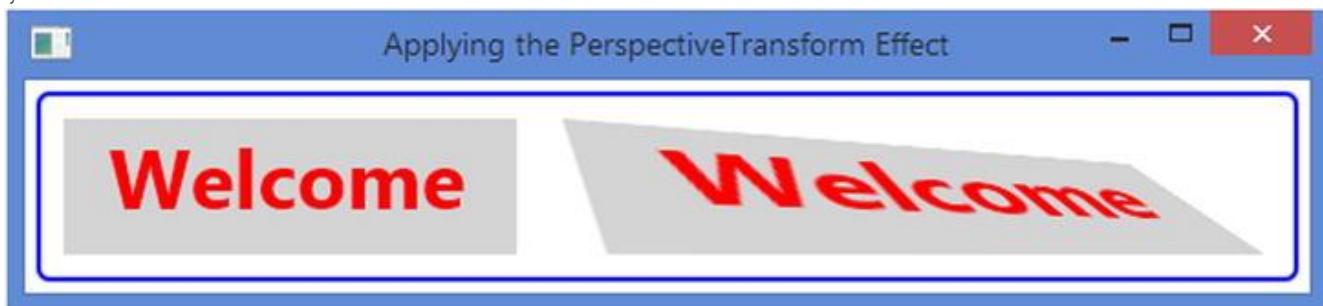
// Group the original nodes
Group group1 = new Group(rect1, text1);

// Group the nodes with the effect
Group group2 = new Group(rect2, text2);
group2.setEffect(effect);
group2.setCache(true); // A hint to cache the bitmap
for the group

HBox root = new HBox(group1, group2);
root.setSpacing(20);
root.setStyle("-fx-padding: 10;" +
             "-fx-border-style: solid inside;" +
             "-fx-border-width: 2;" +
             "-fx-border-insets: 5;" +
             "-fx-border-radius: 5;" +
             "-fx-border-color: blue;");

Scene scene = new Scene(root, 600, 100);
stage.setScene(scene);
stage.setTitle("Applying the PerspectiveTransform
Effect");
stage.show();
}
}
}

```



**Figure 20-35.** Text and Rectangle nodes with a PerspectiveTransform effect

## Summary

An effect is a filter that accepts one or more graphical inputs, applies an algorithm on the inputs, and produces an output. Typically, effects are applied to nodes to create visually appealing user interfaces. Examples of effects are shadow, blur, warp, glow, reflection, blending, and different types of lighting. The JavaFX library provides several effect-related classes. Effect is a conditional feature. Effects applied to nodes will be

ignored if it is not available on a platform. The `Node` class contains an `effect` property that specifies the effect applied to the node. By default, it is `null`. An instance of the `Effect` class represents an effect. The `Effect` class is the abstract base for all effect classes. All effect classes are included in the `javafx.scene.effect` package.

Some effects can be chained with other effects. The effects are applied in sequence. The output of the first effect becomes the input for the second effect and so on. Effect classes that allow chaining contain an `input` property to specify the effect that precedes it. If the `input` property is `null`, the effect is applied to the node on which this effect is set. By default, the `input` property is `null`.

A shadowing effect draws a shadow and applies it to an input. JavaFX supports three types of shadowing effects: `DropShadow`, `InnerShadow`, and `Shadow`.

A blurring effect produces a blurred version of an input. JavaFX lets you apply different types of blurring effects, which differ in the algorithms they use to create the effect. Three types of blurring effects are `BoxBlur`, `GaussianBlur`, and `MotionBlur`.

The `Bloom` effect adds glow to the pixels of its input that have a luminosity greater than or equal to a specified limit. Note that not all pixels in a `Bloom` effect are made to glow. An instance of the `Bloom` class represents a `Bloom` effect.

The `Glow` effect makes the bright pixels of the input brighter. An instance of the `Glow` class represents a `Glow` effect.

The `Reflection` effect adds a reflection of the input below the input. An instance of the `Reflection` class represents a `reflection` effect.

`Sepia` is a reddish-brown color. Sepia toning is performed on black-and-white photographic prints to give them a warmer tone. An instance of the `SepiaTone` class represents a `SepiaTone` effect.

The `DisplacementMap` effect shifts each pixel in the input to produce an output. The name has two parts: `Displacement` and `Map`. The first part implies that the effect displaces the pixels in the input. The second part implies that the displacement is based on a map that provides a displacement factor for each pixel in the output. An instance of the `DisplacementMap` class represents a `DisplacementMap`.

The `ColorInput` effect is a simple effect that fills (floods) a rectangular region with a specified paint. Typically, it is used as an input to another effect. An instance of the `ColorInput` class represents the `ColorInput` effect.

The `ImageInput` effect works like the `ColorInput` effect. It passes the given image as an input to another effect. The given image is not

modified by this effect. Typically, it is used as an input to another effect, not as an effect directly applied to a node. An instance of the `ImageInput` class represents the `ImageInput` effect.

Blending combines two pixels at the same location from two inputs to produce one composite pixel in the output. The `Blend` effect takes two input effects and blends the overlapping pixels of the inputs to produce an output. The blending of two inputs is controlled by a blending mode. JavaFX provides 17 predefined blending modes. An instance of the `Blend` class represents the `Blend` effect.

The `Lighting` effect, as the name suggests, simulates a light source shining on a specified node in a scene to give the node a 3D look. A `Lighting` effect uses a light source, which is an instance of the `Light` class, to produce the effect.

A `PerspectiveTransform` effect gives a 2D node a 3D look by mapping the corners to different locations. The straight lines in the original nodes remain straight. However, parallel lines in the original nodes may not necessarily remain parallel. An instance of the `PerspectiveTransform` class represents a `PerspectiveTransform` effect.

The next chapter will discuss how to apply different types of transformations to nodes.

## CHAPTER 21



### Understanding Transformations

In this chapter, you will learn:

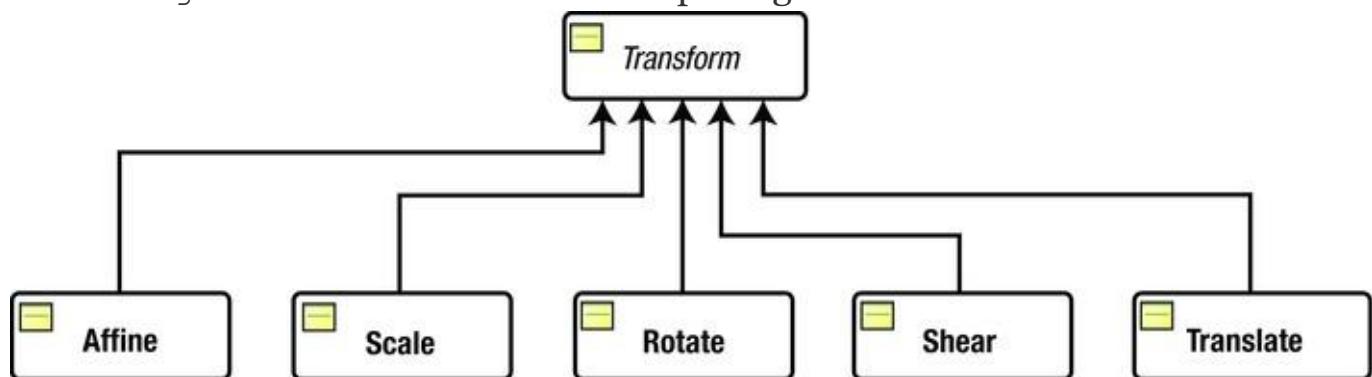
- What a transformation is
- What are translation, rotation, scale, and shear transformations and how to apply them to nodes
- How to apply multiple transformations to a node

#### What Is a Transformation?

A transformation is a mapping of points in a coordinate space to themselves preserving distances and directions between them. Several types of transformations can be applied to points in a coordinate space. JavaFX supports the following types of transformation:

- Translation
- Rotation
- Shear
- Scale
- Affine

An instance of the abstract `Transform` class represents a transformation in JavaFX. The `Transform` class contains common methods and properties used by all types of transformations on nodes. It contains factory methods to create specific types of transformations. Figure 21-1 shows a class diagram for the classes representing different types of transformations. The name of the class matches with the type of transformation the class provides. All classes are in the `javafx.scene.transform` package.



**Figure 21-1.** A class diagram for transform-related classes

An affine transformation is the generalized transformation that preserves the points, straight lines, and planes. The parallel lines remain parallel after the transformation. It may not preserve the angles between lines or the distances between points. However, the ratios of distances between points on a straight line are preserved. Translation, scale, homothetic transformation, similarity transformation, reflection, rotation, shear, and so on are examples of the affine transformation.

An instance of the `Affine` class represents an affine transformation. The class is not easy to use for beginners. Its use requires advanced knowledge of mathematics such as matrix. If you need a specific type of transformation, use the specific subclasses such as `Translate`, `Shear`, and so on, rather than using the generalized `Affine` class. You can also combine multiple individual transformations to create a more complex one. We will not discuss this class in this book.

Using transformations is easy. However, sometimes it is confusing because there are multiple ways to create and apply them.

There are two ways to create a `Transform` instance.

- Use one of the factory methods of the `Transform` class—for example, the `translate()` method for creating a `Translate` object, the `rotate()` method to create a `Rotate` object, etc.
- Use the specific class to create a specific type of transform—for example, the `Translate` class for a translation, the `Rotate` class for a rotation, etc.

Both of the following `Translate` objects represent the same translation:

```
double tx = 20.0;
double ty = 10.0;

// Using the factory method in the Transform class
Translate translate1 = Transform.translate(tx, ty);

// Using the Translate class constructor
Translate translate2 = new Translate(tx, ty);
```

There are two ways to apply a transformation to a node.

- Use the specific properties in the `Node` class. For example, use the `translateX`, `translateY`, and `translateZ` properties of the `Node` class to apply a

translation to a node. Note that you cannot apply a shear transformation this way.

- Use the `transforms` sequence of a node.

The `getTransforms()` method of the `Node` class returns an `ObservableList<Transform>`. Populate this list with all the `Transform` objects. The `Transforms` will be applied in sequence. You can apply a shear transformation only using this method.

The two methods of applying `Transforms` work little differently. We will discuss the differences when we discuss the specific types of transformation. Sometimes, it is possible to use both of the foregoing methods to apply transformations, and in that case, the transformations in the `transforms` sequence are applied before the transformation set on the properties of the node.

The following snippet of code applies three transformations to a rectangle: shear, scale, and translation:

```
Rectangle rect = new Rectangle(100, 50, Color.LIGHTGRAY);
// Apply transforms using the transforms sequence of the
// Rectangle
Transform shear = Transform.shear(2.0, 1.2);
Transform scale = Transform.scale(1.1, 1.2);
rect.getTransforms().addAll(shear, scale);
// Apply a translation using the translateX and translateY
// properties of the Node class
rect.setTranslateX(10);
rect.setTranslateY(10);
```

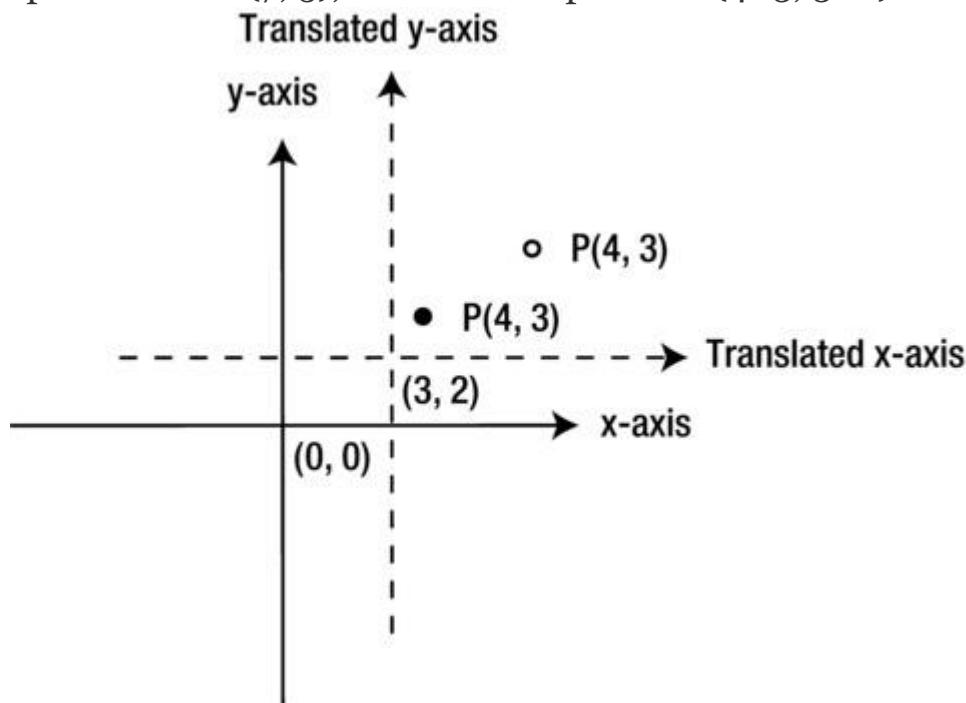
The shear and scale are applied using the `transforms` sequence. The translation is applied using the `translateX` and `translateY` properties of the `Node` class. The transformations in the `transforms` sequence, shear and scale, are applied in sequence followed by the translation.

## The Translation Transformation

A translation moves every point of a node by a fixed distance in a specified direction relative to its parent coordinate system. It is achieved by shifting the origin of the local coordinate system of the node to a new location. Computing the new locations of points is easy—just add a triplet of numbers to the coordinates of each point in a 3D space. In a 2D space, add a pair of numbers to the coordinates of each point.

Suppose you want to apply translation to a 3D coordinate space by  $(tx, ty, tz)$ . If a point had coordinates  $(x, y, z)$  before the translation, after the translation its coordinates would be  $(x + tx, y + ty, z + tz)$ .

Figure 21-2 shows an example of a translation transformation. Axes before the transformations are shown in solid lines. Axes after the transformations are shown in dashed lines. Note that the coordinates of the point P remains the same (4, 3) in the translated coordinate spaces. However, the coordinates of the point relative to the original coordinate space change after the transformation. The point in the original coordinate space is shown in a solid black fill color, and in the transformed coordinate space, it is shown without a fill color. The origin of the coordinate system (0, 0) has been shifted to (3, 2). The coordinates of the point P (the shifted point) in the original coordinate space become (7, 5), which is computed as (4+3, 3+2).



**Figure 21-2.** An example of a translation transformation

An instance of the `Translate` class represents a translation. It contains three properties.

- `x`
- `y`
- `z`

The properties specify the x, y, and z coordinates of the new origin of the local coordinate system of the node after translation. The default values for the properties are 0.0.

The `Translate` class provides three constructors.

- `Translate()`

- `Translate(double x, double y)`
- `Translate(double x, double y, double z)`

The no-args constructor creates a `Translate` object with the default values for the `x`, `y`, and `z` properties, which, in essence, represents no translation. The other two constructors let you specify the translation distance along the three axes. A transformation to a `Group` is applied to all the nodes in the `Group`.

Compare the use of the `layoutX` and `layoutY` properties of the `Node` class with the `translateX` and `translateY` properties. The `layoutX` and `layoutY` properties position the node in its local coordinate system without transforming the local coordinate system whereas the `translateX` and `translateY` properties transform the local coordinate system of the node by shifting the origin. Typically, `layoutX` and `layoutY` are used to place a node in a scene whereas translation is used for moving a node in an animation. If you set both properties for a node, its local coordinate system will be transformed using the translation, and, then, the node will be placed in the new coordinate system using its `layoutX` and `layoutY` properties.

The program in Listing 21-1 creates three rectangles. By default, they are placed at  $(0, 0)$ . It applies a translation to the second and third rectangles. Figure 21-3 shows the rectangles after the translation.

### ***Listing 21-1.*** Applying Translations to Nodes

```
// TranslateTest.java
package com.jdojo.transform;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.transform.Translate;
import javafx.stage.Stage;

public class TranslateTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Rectangle rect1 = new Rectangle(100, 50,
Color.LIGHTGRAY);
        rect1.setStroke(Color.BLACK);

        Rectangle rect2 = new Rectangle(100, 50,
```

```

        Color.YELLOW);
        rect2.setStroke(Color.BLACK);

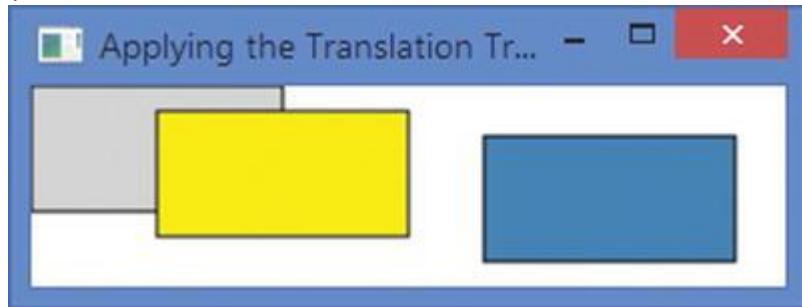
        Rectangle rect3 = new Rectangle(100, 50,
Color.STEELBLUE);
        rect3.setStroke(Color.BLACK);

        // Apply a translation on rect2 using the transforms
sequence
        Translate translate1 = new Translate(50, 10);
        rect2.getTransforms().addAll(translate1);

        // Apply a translation on rect3 using the translateX
// and translateY properties
        rect3.setTranslateX(180);
        rect3.setTranslateY(20);

        Pane root = new Pane(rect1, rect2, rect3);
        root.setPrefSize(300, 80);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Applying the Translation
Transformation");
        stage.show();
    }
}

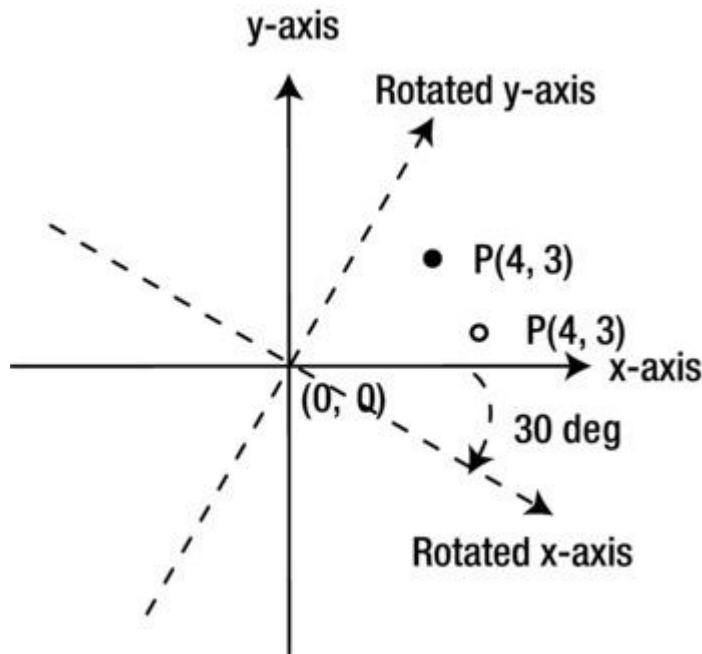
```



**Figure 21-3.** Rectangles with translations

## The Rotation Transformation

In a rotation transformation, the axes are rotated around a pivot point in the coordinate space and the coordinates of points are mapped to the new axes. Figure 21-4 shows the axes of a coordinate system in a 2D plane rotated by an angle of 30 degrees. The axis of rotation is z-axis. The origin of the original coordinate system is used as the pivot point of rotation. The original axes are shown in solid lines and the rotated axes in dashed lines. The point P in the original coordinate system is shown in a black fill and in the rotated coordinate system with no fill.



**Figure 21-4.** An example of a rotation transformation

An instance of the `Rotate` class represents a rotation transformation. It contains five properties to describe the rotation:

- `angle`
- `axis`
- `pivotX`
- `pivotY`
- `pivotZ`

The `angle` property specifies the angle of rotation in degrees. The default is 0.0 degrees. A positive value for the `angle` is measured clockwise.

The `axis` property specifies the axis of rotation at the pivot point. Its value can be one of the constants, `X_AXIS`, `Y_AXIS`, and `Z_AXIS`, defined in the `Rotate` class. The default axis of rotation is `Rotate.Z_AXIS`.

The `pivotX`, `pivotY`, and `pivotZ` properties are the x, y, and z coordinates of the pivot point. The default values for the properties are 0.0.

The `Rotate` class contains several constructors:

- `Rotate()`
- `Rotate(double angle)`
- `Rotate(double angle, double pivotX, double pivotY)`

- `Rotate(double angle, double pivotX, double pivotY, double pivotZ)`
- `Rotate(double angle, double pivotX, double pivotY, double pivotZ, Point3D axis)`
- `Rotate(double angle, Point3D axis)`

The no-args constructor creates an identity rotation, which does not have any effect on the transformed node. The other constructors let you specify the details.

The program in Listing 21-2 creates two rectangles and places them at the same location. The opacity of the second rectangle is set to 0.5, so we can see through it. The coordinate system of the second rectangle is rotated by 30 degrees in the clockwise direction using the origin as the pivot point. Figure 21-5 shows the rotated rectangle.

### ***Listing 21-2.*** Using a Rotation Transformation

```
// RotateTest.java
package com.jdojo.transform;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.transform.Rotate;
import javafx.stage.Stage;

public class RotateTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

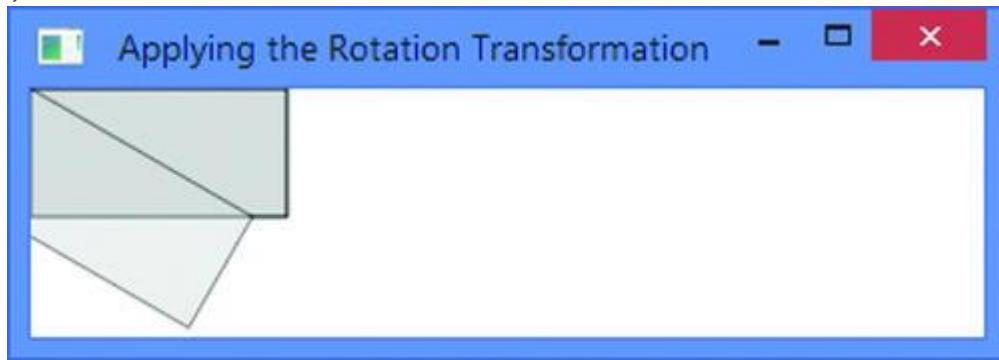
    @Override
    public void start(Stage stage) {
        Rectangle rect1 = new Rectangle(100, 50,
Color.LIGHTGRAY);
        rect1.setStroke(Color.BLACK);

        Rectangle rect2 = new Rectangle(100, 50,
Color.LIGHTGRAY);
        rect2.setStroke(Color.BLACK);
        rect2.setOpacity(0.5);

        // Apply a rotation on rect2. The rotation angle is
        // 30 degree clockwise
        // (0, 0) is the pivot point
        Rotate rotate = new Rotate(30, 0, 0);
        rect2.getTransforms().addAll(rotate);

        Pane root = new Pane(rect1, rect2);
```

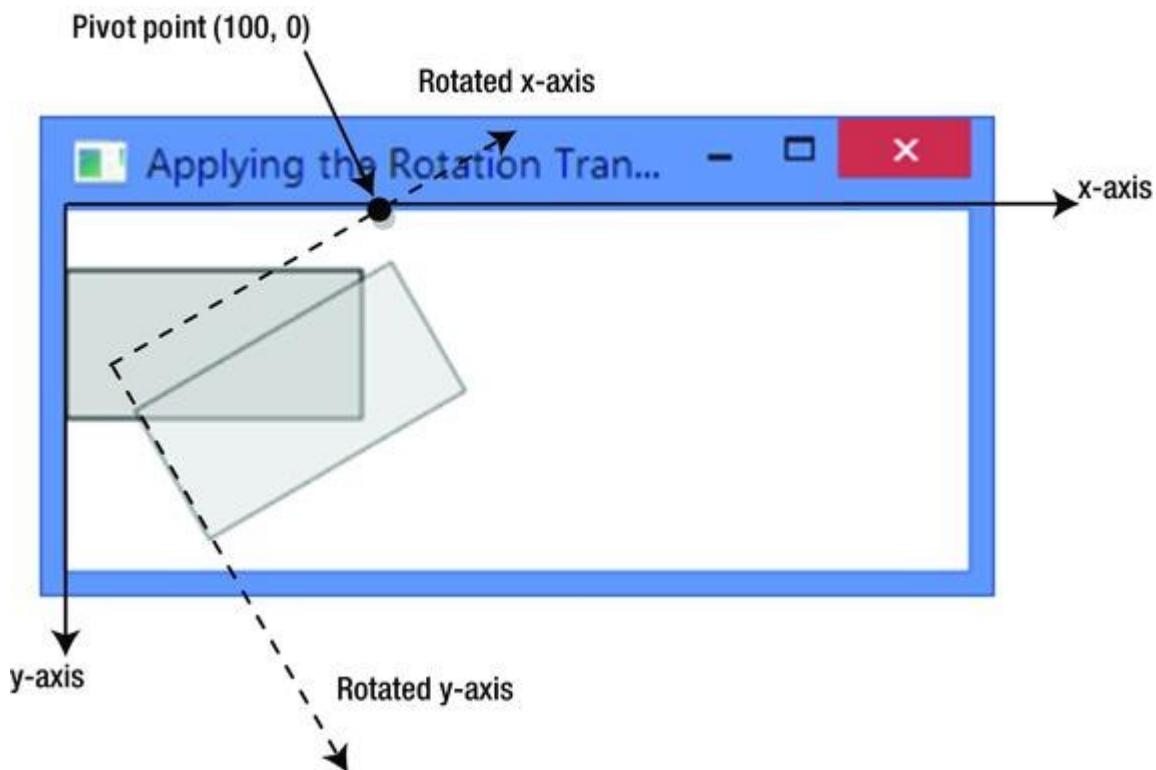
```
        root.setPrefSize(300, 80);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Applying the Rotation
Transformation");
        stage.show();
    }
}
```



**Figure 21-5.** A Rectangle using a rotation transformation

It is easy to visualize the effect of a rotation when the pivot point is the origin of the local coordinate system of the node and the upper-left corner of a node is located at the origin as well. Let us consider the following snippet of code that rotates a rectangle as shown in Figure 21-6.

```
Rectangle rect1 = new Rectangle(100, 50, Color.LIGHTGRAY);
rect1.setY(20);
rect1.setStroke(Color.BLACK);
Rectangle rect2 = new Rectangle(100, 50, Color.LIGHTGRAY);
rect2.setY(20);
rect2.setStroke(Color.BLACK);
rect2.setOpacity(0.5);
// Apply a rotation on rect2. The rotation angle is 30 degree
anticlockwise
// (100, 0) is the pivot point.
Rotate rotate = new Rotate(-30, 100, 0);
rect2.getTransforms().addAll(rotate);
```



**Figure 21-6.** Rotating a Rectangle using a pivot point other than the origin of its local coordinate system

The coordinates of the upper-left of the rectangles are set to (0, 20). A point at (100, 0) is used as the pivot point to rotate the second rectangle. The pivot point is located on the x-axis of the rectangle. The coordinate system of the second rectangle is pinned at (100, 0), and then, rotated by 30 degree in the anticlockwise direction. Notice that the second rectangle maintains its location (0, 20) in the rotated coordinate space.

You can also apply a rotation to a node using the `rotate` and `rotationAxis` properties of the `Node` class. The `rotate` property specifies the angle of rotation in degrees. The `rotationAxis` property specifies the axis of rotation. The center of the untransformed layout bounds of the node is used as the pivot point.

**Tip** The default pivot point used in a `transforms` sequence is the origin of the local coordinate system of the node whereas the `rotate` property of the `Node` class uses the center of the untransformed layout bounds of the node as the pivot point.

The program in Listing 21-3 creates two rectangles similar to the ones in Listing 21-2. It uses the `rotate` property of the `Node` class to rotate the rectangle by 30 degrees. Figure 21-7 shows the rotated rectangle. Compare the rotated rectangles in Figure 21-5 and Figure 21-7. The former uses the origin of the local coordinate system as the pivot point and the latter uses the center of the rectangle as the pivot point.

**Listing 21-3.** Using the rotate Property of the Node Class to Rotate a Rectangle

```
// RotatePropertyTest.java
package com.jdojo.transform;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

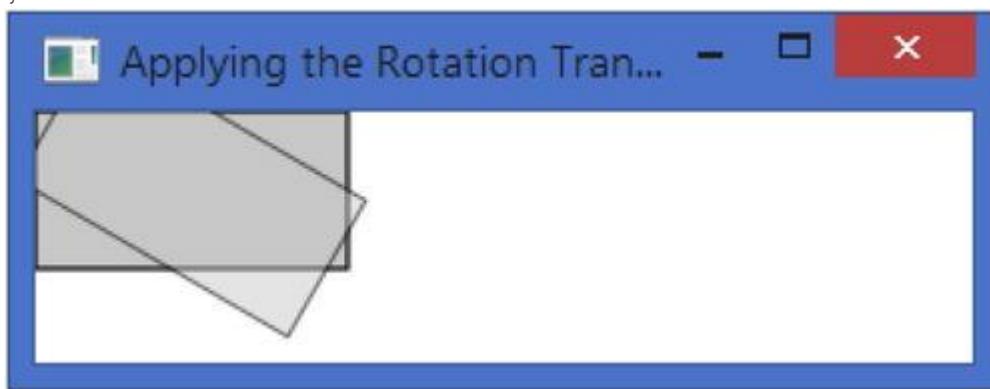
public class RotatePropertyTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Rectangle rect1 = new Rectangle(100, 50,
Color.LIGHTGRAY);
        rect1.setStroke(Color.BLACK);

        Rectangle rect2 = new Rectangle(100, 50,
Color.LIGHTGRAY);
        rect2.setStroke(Color.BLACK);
        rect2.setOpacity(0.5);

        // Use the rotate proeprty of the node class
        rect2.setRotate(30);

        Pane root = new Pane(rect1, rect2);
        root.setPrefSize(300, 80);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Applying the Rotation
Transformation");
        stage.show();
    }
}
```



**Figure 21-7.** A Rectangle rotated using the rotate property of the Node class

## The Scale Transformation

A scale transformation scales the unit of measurement along axes of a coordinate system by a scale factor. This causes the dimensions of a node to change (stretch or shrink) by the specified scale factors along axes. The dimension along an axis is multiplied by the scale factor along that axis. The transformation is applied at a pivot point whose coordinates remain the same after the transformation.

An instance of the `Scale` class represents a scale transformation. It contains the following six properties to describe the transformation:

- `x`
- `y`
- `z`
- `pivotX`
- `pivotY`
- `pivotZ`

The `x`, `y`, and `z` properties specify the scale factors long the `x`-axis, `y`-axis, and `z`-axis. They are `1.0` by default.

The `pivotX`, `pivotY`, and `pivotZ` properties are the `x`, `y`, and `z` coordinates of the pivot point. The default values for the properties are `0.0`.

The `Scale` class contains several constructors.

- `Scale()`
- `Scale(double x, double y)`
- `Scale(double x, double y, double z)`
- `Scale(double x, double y, double pivotX, double pivotY)`
- `Scale(double x, double y, double z, double pivotX, double pivotY, double pivotZ)`

The no-args constructor creates an identity scale transformation, which does not have any effect on the transformed node. The other constructors let you specify the scale factors and the pivot point.

You can use an object of the `Scale` class or the `scaleX`, `scaleY`, and `scaleZ` properties of the `Node` class to apply a scale transformation. By default, the pivot point used by the `Scale` class is at `(0, 0, 0)`. The properties of the `Node` class use the center of the node as the pivot point.

The program in Listing 21-4 creates two rectangles. Both are placed at the same location. One of them is scaled and the other not. The opacity of the not scaled rectangle is set to `0.5`, so we can see through it. Figure

**21-8** shows the rectangles. The scaled rectangle is smaller. The coordinate system of the second rectangle is scaled by 0.5 along the x-axis and 0.50 along the y-axis. The `scaleX` and `scaleY` properties are used to apply the transformation, which uses the center of the rectangles as the pivot point making the rectangles shrunk, but keeping it at the same location.

#### ***Listing 21-4.*** Using Scale Transformations

```
// ScaleTest.java
package com.jdojo.transform;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

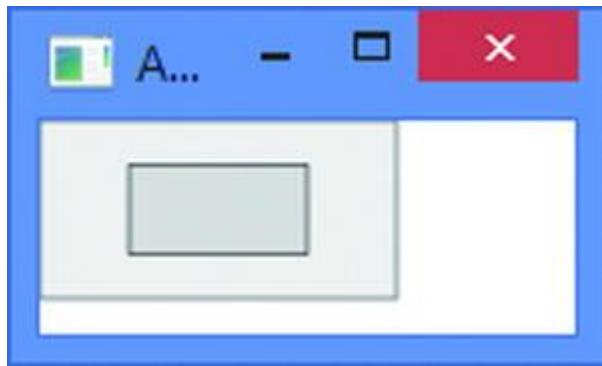
public class ScaleTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Rectangle rect1 = new Rectangle(100, 50,
Color.LIGHTGRAY);
        rect1.setStroke(Color.BLACK);
        rect1.setOpacity(0.5);

        Rectangle rect2 = new Rectangle(100, 50,
Color.LIGHTGRAY);
        rect2.setStroke(Color.BLACK);

        // Apply a scale on rect2. Center of the Rectangle
        // is the pivot point.
        rect2.setScaleX(0.5);
        rect2.setScaleY(0.5);

        Pane root = new Pane(rect1, rect2);
        root.setPrefSize(150, 60);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Applying the Scale Transformation");
        stage.show();
    }
}
```



**Figure 21-8.** Two Rectangles using scale transformations

If the pivot point is not the center of the node, the scale transformation may move the node. The program in Listing 21-5 creates two rectangles. Both are placed at the same location. One of them is scaled and the other not. The opacity of the not scaled rectangle is set to 0.5, so we can see through it. Figure 21-9 shows the rectangles. The scaled rectangle is smaller. A `Scale` object with the `transforms` sequence is used to apply the transformation, which uses the upper-left corner of the rectangle as the pivot point making the rectangle shrink, but moving it to the left to keep the coordinates of its upper-left corner the same (150, 0) in the transformed coordinate system. The scaled rectangle shrinks by half (scale factor = 0.50) in both directions and moves half the distance to the left.

### **Listing 21-5.** Using Scale Transformations

```
// ScalePivotPointTest.java
package com.jdojo.transform;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;

import javafx.scene.transform.Scale;
import javafx.stage.Stage;

public class ScalePivotPointTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Rectangle rect1 = new Rectangle(100, 50,
Color.LIGHTGRAY);
        rect1.setX(150);
        rect1.setStroke(Color.BLACK);
        stage.setScene(new Scene(rect1));
        stage.show();
    }
}
```

```

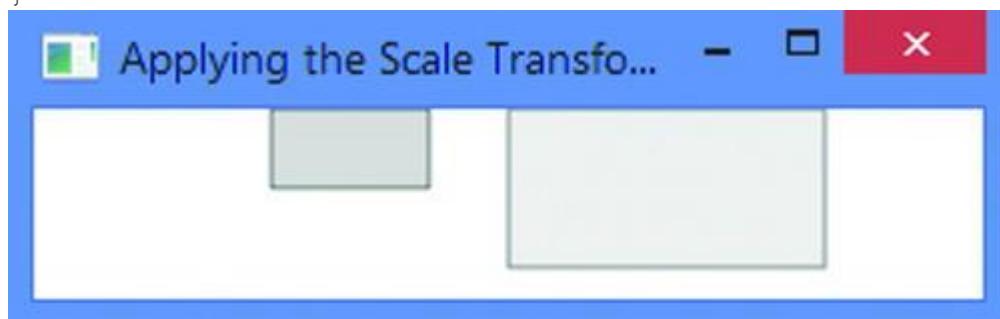
        rect1.setOpacity(0.5);

        Rectangle rect2 = new Rectangle(100, 50,
Color.LIGHTGRAY);
        rect2.setX(150);
        rect2.setStroke(Color.BLACK);

        // Apply a scale on rect2. The origin of the local
coordinate system
        // of rect4 is the pivot point
        Scale scale = new Scale(0.5, 0.5);
        rect2.getTransforms().addAll(scale);

        Pane root = new Pane(rect1, rect2);
        root.setPrefSize(300, 60);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Applying the Scale Transformation");
        stage.show();
    }
}

```



**Figure 21-9.** Two Rectangles using scale transformations

## The Shear Transformation

A shear transformation rotates axes of the local coordinate system of the node around a pivot point, so the axes are no longer perpendicular. A rectangular node becomes a parallelogram after the transformation.

An instance of the `Shear` class represents a shear transformation. It contains four properties to describe the transformation.

- `x`
- `y`
- `pivotX`
- `pivotY`

The `x` property specifies a multiplier by which the coordinates of points are shifted along the positive x-axis by a factor of the y coordinate of the point. The default is 0.0.

The `y` property specifies a multiplier by which the coordinates of points are shifted along the positive `y`-axis by a factor of the `x` coordinate of the point. The default is `0.0`.

The `pivotX`, and `pivotY` properties are the `x` and `y` coordinates of the pivot point about which the shear occurs. The default values for them are `0.0`. The pivot point is not shifted by the shear. By default, the pivot point is the origin of the untransformed coordinate system.

Suppose you have a point  $(x_1, y_1)$  inside a node, and by the shear transformation, the point is shifted to  $(x_2, y_2)$ . You can use the following formula to compute  $(x_2, y_2)$ :

$$\begin{aligned}x_2 &= \text{pivotX} + (x_1 - \text{pivotX}) + x * (y_1 - \text{pivotY}) \\y_2 &= \text{pivotY} + (y_1 - \text{pivotY}) + y * (x_1 - \text{pivotX})\end{aligned}$$

All coordinates ( $x_1$ ,  $y_1$ ,  $x_2$ , and  $y_2$ ) in the previous formula are in the untransformed local coordinate system of the node. Notice that if  $(x_1, y_1)$  is the pivot point, the foregoing formula computes the shifted point  $(x_2, y_2)$ , which is the same as  $(x_1, y_1)$ . That is, the pivot point is not shifted.

The `Shear` class contains several constructors.

- `Shear()`
- `Shear(double x, double y)`
- `Shear(double x, double y, double pivotX, double pivotY)`

The no-args constructor creates an identity shear transformation, which does not have any effect on the transformed node. The other constructors let you specify the shear multipliers and the pivot point.

**Tip** You can apply a shear transformation to a node using only a `Shear` object in the `transforms` sequence. Unlike for other types of transformations, the `Node` class does not contain a property allowing you to apply shear transformation.

The program in Listing 21-6 applies a `Shear` to a rectangle as shown in Figure 21-10. The original rectangle is also shown. A multiplier of `0.5` is used along both axes. Note that the pivot point is  $(0, 0)$ , which is the default.

### ***Listing 21-6.*** Using the Shear Transformation

```
// ShearTest.java
package com.jdojo.transform;

import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
```

```

import javafx.scene.shape.Rectangle;
import javafx.scene.transform.Shear;
import javafx.stage.Stage;

public class ShearTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

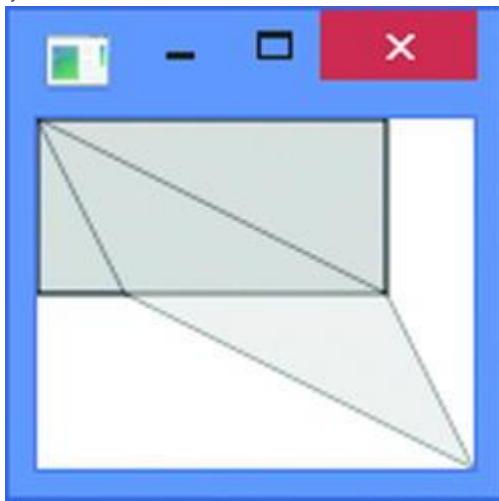
    @Override
    public void start(Stage stage) {
        Rectangle rect1 = new Rectangle(100, 50,
Color.LIGHTGRAY);
        rect1.setStroke(Color.BLACK);

        Rectangle rect2 = new Rectangle(100, 50,
Color.LIGHTGRAY);
        rect2.setStroke(Color.BLACK);
        rect2.setOpacity(0.5);

        // Apply a shear on rect2. The x and y multipliers
are 0.5 and
        // (0, 0) is the pivot point.
        Shear shear = new Shear(0.5, 0.5);
        rect2.getTransforms().addAll(shear);

        Group root = new Group(rect1, rect2);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Applying the Shear Transformation");
        stage.show();
    }
}

```



**Figure 21-10.** A Rectangle with a shear transformation using (0, 0) as the pivot point

Let us use a pivot point other than (0, 0) for a Shear transformation. Consider the following snippet of code:

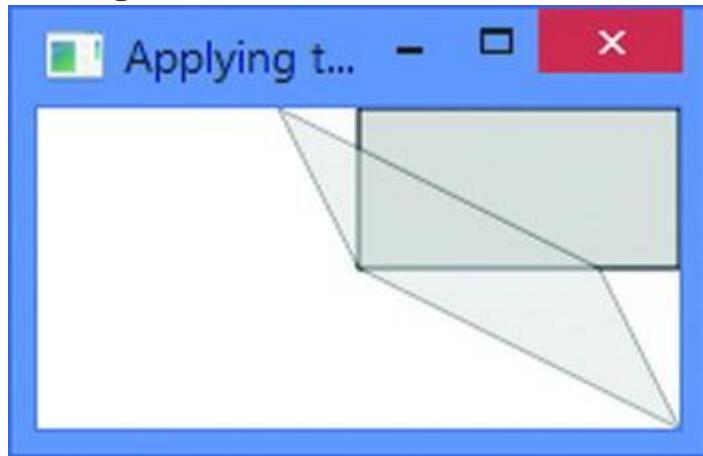
```

Rectangle rect1 = new Rectangle(100, 50, Color.LIGHTGRAY);
rect1.setX(100);
rect1.setStroke(Color.BLACK);
Rectangle rect2 = new Rectangle(100, 50, Color.LIGHTGRAY);
rect2.setX(100);
rect2.setStroke(Color.BLACK);
rect2.setOpacity(0.5);

// Apply a shear on rect2. The x and y multipliers are 0.5 and
// (100, 50) is the pivot point.
Shear shear = new Shear(0.5, 0.5, 100, 50);
rect2.getTransforms().addAll(shear);

```

The code is similar to the one shown in Listing 21-6. The upper-left corners of the rectangles are placed at (100, 0), so we can see the sheared rectangle fully. We have used (100, 50), which is the lower-left corner of the rectangle, as the pivot point. Figure 21-11 shows the transformed rectangle. Notice that the transformation did not shift the pivot point.



**Figure 21-11.** A Rectangle with a shear transformation using (100, 50) as the pivot point

Let us apply our formula to validate the coordinates of the upper-right corner, which is originally at (200, 0) relative to the untransformed coordinate system of the rectangle.

```

x1 = 200
y1 = 0
pivotX = 100
pivotY = 50
x = 0.5
y = 0.5

x2 = pivotX + (x1 - pivotX) + x * (y1 - pivotY)
    = 100 + (200 - 100) + 0.5 * (0 - 50)
    = 175

y2 = pivotY + (y1 - pivotY) + y * (x1 - pivotX)
    = 50 + (0 - 50) + 0.5 * (200 - 100)
    = 50

```

Therefore, (175, 50) is the shifted location of the upper-right corner in the untransformed coordinate system of the rectangle.

## Applying Multiple Transformations

You can apply multiple transformations to a node. As mentioned previously, the transformations in the `transforms` sequence are applied before the transformation set on the properties of the node. When properties of the `Node` class are used, translation, rotation, and scale are applied in sequence. When the `transforms` sequence is used, transformations are applied in the order they are stored in the sequence.

The program in Listing 21-7 creates three rectangles and positions them at the same location. It applies multiple transformations to the second and third rectangles in different order. Figure 21-12 shows the result. The first rectangle is shown at its original position, as we did not apply any transformation to it. Notice that two rectangles ended up at different locations. If you change the order of the transformation for the third rectangle as shown next, both rectangles will overlap.

```
rect3.getTransforms().addAll(new Translate(100, 0),
    new Rotate(30, 50, 25),
    new Scale(1.2, 1.2, 50, 25));
```

### **Listing 21-7.** Using Multiple Transformations on a Node

```
// MultipleTransformations.java
package com.jdojo.transform;

import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.transform.Rotate;
import javafx.scene.transform.Scale;
import javafx.scene.transform.Translate;
import javafx.stage.Stage;

public class MultipleTransformations extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Rectangle rect1 = new Rectangle(100, 50,
Color.LIGHTGRAY);
        rect1.setStroke(Color.BLACK);

        Rectangle rect2 = new Rectangle(100, 50,
```

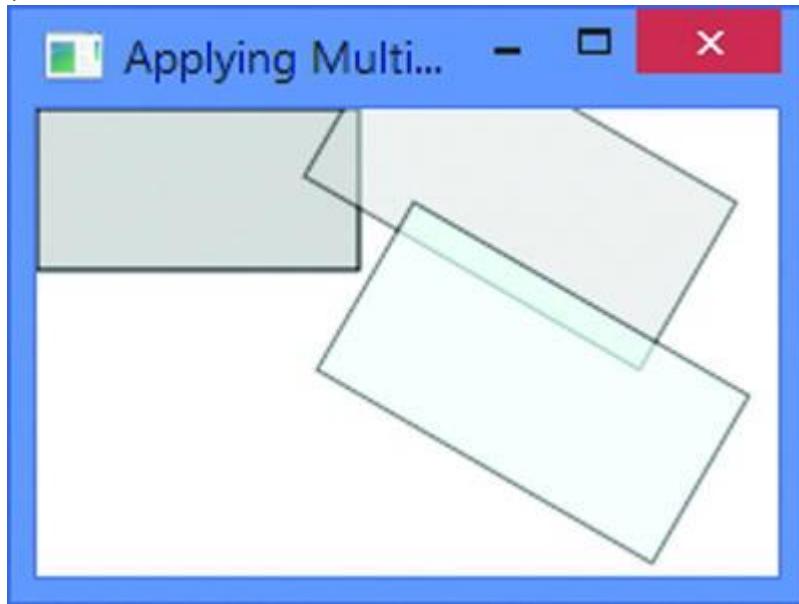
```
Color.LIGHTGRAY);
    rect2.setStroke(Color.BLACK);
    rect2.setOpacity(0.5);

    Rectangle rect3 = new Rectangle(100, 50,
Color.LIGHTCYAN);
    rect3.setStroke(Color.BLACK);
    rect3.setOpacity(0.5);

    // apply transformations to rect2
    rect2.setTranslateX(100);
    rect2.setTranslateY(0);
    rect2.setRotate(30);
    rect2.setScaleX(1.2);
    rect2.setScaleY(1.2);

    // Apply the same transformation as on rect2, but in
a different order
    rect3.getTransforms().addAll(new Scale(1.2, 1.2, 50,
25),
                                new Rotate(30, 50, 25),
                                new Translate(100, 0));

    Group root = new Group(rect1, rect2, rect3);
    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("Applying Multiple Transformations");
    stage.show();
}
}
```



**Figure 21-12.** Rectangles with multiple transformations

## Summary

A transformation is a mapping of points in a coordinate space to themselves preserving distances and directions between them. Several types of transformations can be applied to points in a coordinate space. JavaFX supports the following types of transformation: translation, rotation, shear, scale, and affine.

An instance of the abstract `Transform` class represents a transformation in JavaFX. The `Transform` class contains common methods and properties used by all types of transformations on nodes. It contains factory methods to create specific types of transformations. All transformation classes are in the `javafx.scene.transform` package.

An affine transformation is the generalized transformation that preserves the points, lines, and planes. The parallel lines remain parallel after the transformation. The affine transformation may not preserve the angles between lines and the distances between points. However, the ratios of distances between points on a straight line are preserved. Translation, scale, homothetic transformation, similarity transformation, reflection, rotation, and shear are examples of the affine transformation. An instance of the `Affine` class represents an affine transformation.

There are two ways to apply a transformation to a node: using the specific properties in the `Node` class and using the `transforms` sequence of a node.

A translation moves every point of a node by a fixed distance in a specified direction relative to its parent coordinate system. It is achieved by shifting the origin of the local coordinate system of the node to a new location. An instance of the `Translate` class represents a translation.

In a rotation transformation, the axes are rotated around a pivot point in the coordinate space and the coordinates of points are mapped to the new axes. An instance of the `Rotate` class represents a rotation transformation.

A scale transformation scales the unit of measurement along axes of a coordinate system by a scale factor. This causes the dimensions of a node to change (stretch or shrink) by the specified scale factors along axes. The dimension along an axis is multiplied by the scale factor along that axis. The transformation is applied at a pivot point whose coordinates remain the same after the transformation. An instance of the `Scale` class represents a scale transformation.

A shear transformation rotates axes of the local coordinate system of the node around a pivot point, so the axes are no longer perpendicular. A

rectangular node becomes a parallelogram after the transformation. An instance of the `Shear` class represents a shear transformation.

You can apply multiple transformations to a node. The transformations in the `transforms` sequence are applied before the transformation set on the properties of the node. When properties of the `Node` class are used, translation, rotation, and scale are applied in order. When the `transforms` sequence is used, transformations are applied in the order they are stored in the sequence.

The next chapter will discuss how to apply animation to nodes.

## CHAPTER 22



### Understanding Animation

In this chapter, you will learn:

- What animation is in JavaFX
- About classes in JavaFX that are used in performing animation in JavaFX
- How to perform a timeline animation and how to set up cue points on a timeline animation
- How to control animation such as playing, reversing, pausing, and stopping
- How to perform animation using transitions
- About different types of interpolators and their roles in animation

### What Is Animation?

In real world, *animation* implies some kind of motion, which is generated by displaying images in quick succession. For example, when you watch a movie, you are watching images, which change so quickly that you get an illusion of motion.

In JavaFX, animation is defined as changing the property of a node over time. If the property that changes determines the location of the node, the animation in JavaFX will produce an illusion of motion as found in movies. Not all animations have to involve motion; for example, changing the `fill` property of a shape over time is an animation in JavaFX that does not involve motion.

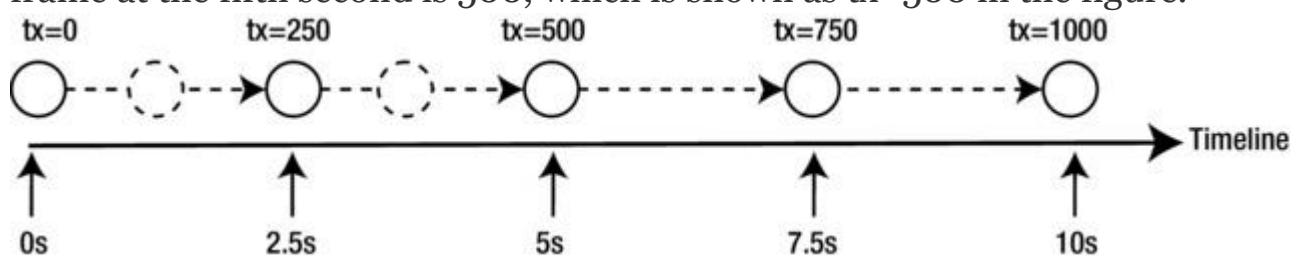
To understand how animation is performed, it is important to understand some key concepts.

- Timeline
- Key frame
- Key value
- Interpolator

Animation is performed over a period of time. A *timeline* denotes the progression of time during animation with an associated key frame at a

given instant. A *key frame* represents the state of the node being animated at a specific instant on the timeline. A key frame has associated key values. A *key value* represents the value of a property of the node along with an interpolator to be used.

Suppose you want to move a circle in a scene from left to right horizontally in 10 seconds. Figure 22-1 shows the circle at some positions. . . . The thick horizontal line represents a timeline. Circles with a solid outline represent the key frames at specific instants on the timeline. The key values associated with key frames are shown at the top line. For example, the value for `translateX` property of the circle for the key frame at the fifth second is 500, which is shown as `tx=500` in the figure.

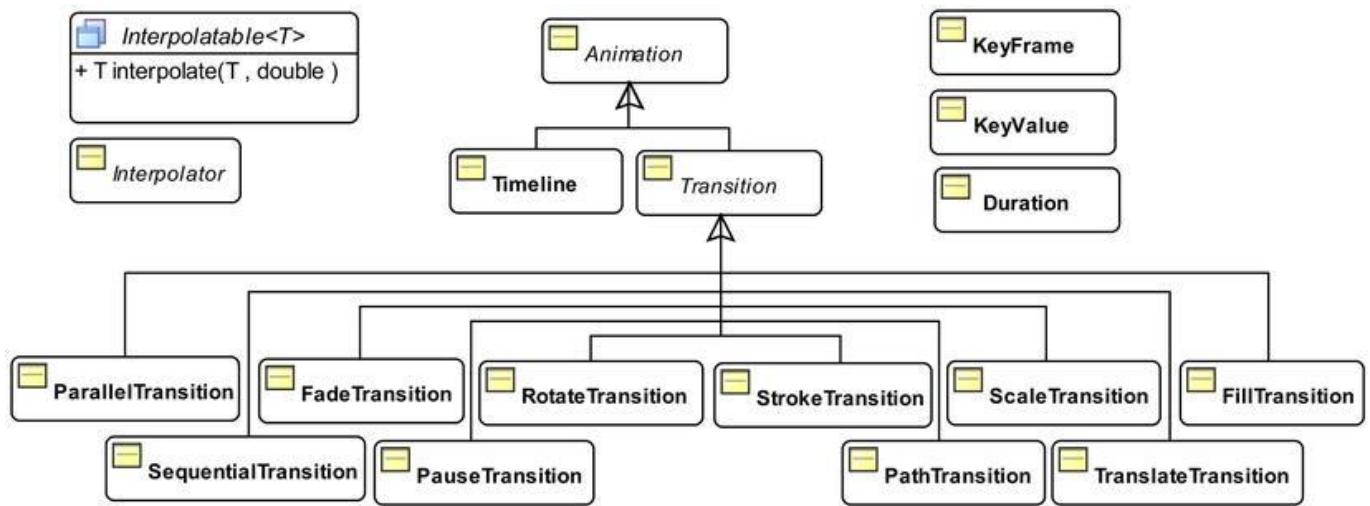


**Figure 22-1.** Animating a circle along a horizontal line using a timeline

The developer provides timelines, key frames, and key values. In this example, there are five key frames. If JavaFX shows only five key frames at the five respective instants, the animation will look jerky. To provide a smooth animation, JavaFX needs to interpolate the position of the circle at any instant on the timeline. That is, JavaFX needs to create intermediate key frames between two consecutive provided key frames. JavaFX does this with the help of an *interpolator*. By default, it uses a *linear interpolator*, which changes the property being animated linearly with time. That is, if the time on the timeline passes  $x\%$ , the value of the property will be  $x\%$  between the initial and final target values. Circles with the dashed outline are created by JavaFX using an interpolator.

## Understanding Animation Classes

Classes providing animation in JavaFX are in the `javafx.animation` package, except the `Duration` class, which is in the `javafx.util` package. Figure 22-2 shows a class diagram for most of the animation-related classes.



**Figure 22-2.** A class diagram for core classes used in animation

The abstract `Animation` class represents an Animation. It contains common properties and methods used by all types of animation.

JavaFX supports two types of animations.

- Timeline animations
- Transitions

In a timeline animation, you create a timeline and add key frames to it. JavaFX creates the intermediate key frames using an interpolator. An instance of the `Timeline` class represents a timeline animation. This type of animation requires a little more code, but it gives you more control.

Several types of animations are commonly performed (moving a node along a path, changing the opacity of a node over time, etc.). These types of animations are known as transitions. They are performed using an internal timeline. An instance of the `Transition` class represents a transition animation. Several subclasses of the `Transition` class exist to support specific types of transitions. For example, the `FadeTransition` class implements a fading effect animation by changing the opacity of a node over time. You create an instance of the `Transition` class (typically, an instance of one of its subclasses), specify the initial and final values for the property to be animated and the duration for the animation. JavaFX takes care of creating the timeline and performing the animation. This type of animation is easier to use.

Sometimes, you may want to perform multiple transitions sequentially or simultaneously.

The `SequentialTransition` and `ParallelTransition` classes let

you perform a set of transitions sequentially and simultaneously, respectively.

## Understanding Utility Classes

Before discussing the details of JavaFX animation, I will discuss a few utility classes that are used in implementing animations. The following sections will discuss those classes.

### Understanding the Duration Class

The `Duration` class is in the `javafx.util` package. It represents a duration of time in milliseconds, seconds, minutes, and hours. It is an immutable class. A `Duration` represents the amount of time for each cycle of an animation. A `Duration` can represent a positive or negative duration.

You can create a `Duration` object in three ways.

- Using the constructor
- Using factory methods
- Using the `valueOf()` method from a duration in String format

The constructor takes the amount of time in milliseconds.

```
Duration tenMillis = new Duration(10);
```

Factory methods create `Duration` objects for different units of time.

They are `millis()`, `seconds()`, `minutes()`, and `hours()`.

```
Duration tenMillis = Duration.millis(10);
Duration tenSeconds = Duration.seconds(10);
Duration tenMinutes = Duration.minutes(10);
Duration tenHours = Duration.hours(10);
```

The `valueOf()` static method takes a String argument containing the duration of time and returns a `Duration` object. The format of the argument is “number [ms | s | m | h]”, where number is the amount of time, and ms, s, m, and h denote milliseconds, seconds, minutes, and hours, respectively.

```
Duration tenMillis = Duration.valueOf("10.0ms");
Duration tenMililsNeg = Duration.valueOf("-10.0ms");
```

You can also represent a duration of an unknown amount of time and an indefinite time using the `UNKNOWN` and `INDEFINITE` constants of the `Duration` class, respectively. You can use the `isIndefinite()` and `isUnknown()` methods to check if a duration represents an indefinite or unknown amount of time. |The class

declares two more constants, ONE and ZERO, that represent durations of 1 millisecond and 0 (no time), respectively.

The Duration class provides several methods to manipulate durations (adding a duration to another duration, dividing and multiplying a duration by a number, comparing two durations, etc.). Listing 22-1 shows how to use the Duration class.

### ***Listing 22-1.*** Using the Duration Class

```
// DurationTest.java
package com.jdojo.animation;

import javafx.util.Duration;

public class DurationTest {
    public static void main(String[] args) {
        Duration d1 = Duration.seconds(30.0);
        Duration d2 = Duration.minutes(1.5);
        Duration d3 = Duration.valueOf("35.25ms");
        System.out.println("d1 = " + d1);
        System.out.println("d2 = " + d2);
        System.out.println("d3 = " + d3);

        System.out.println("d1.toMillis() = "
+ d1.toMillis());
        System.out.println("d1.toSeconds() = "
+ d1.toSeconds());
        System.out.println("d1.toMinutes() = "
+ d1.toMinutes());
        System.out.println("d1.toHours() = "
+ d1.toHours());

        System.out.println("Negation of d1 = "
+ d1.negate());
        System.out.println("d1 + d2 = " + d1.add(d2));
        System.out.println("d1 / 2.0 = " + d1.divide(2.0));

        Duration inf = Duration.millis(1.0/0.0);
        Duration unknown = Duration.millis(0.0/0.0);
        System.out.println("inf.isIndefinite() = "
+ inf.isIndefinite());
        System.out.println("unknown.isUnknown() = "
+ unknown.isUnknown());
    }
}

d1 = 30000.0 ms
d2 = 90000.0 ms
d3 = 35.25 ms
d1.toMillis() = 30000.0
d1.toSeconds() = 30.0
d1.toMinutes() = 0.5
d1.toHours() = 0.008333333333333333
```

```

Negation of d1 = -30000.0 ms
d1 + d2 = 120000.0 ms
d1 / 2.0 = 15000.0 ms
inf.isIndefinite() = true
unknown.isUnknown() = true

```

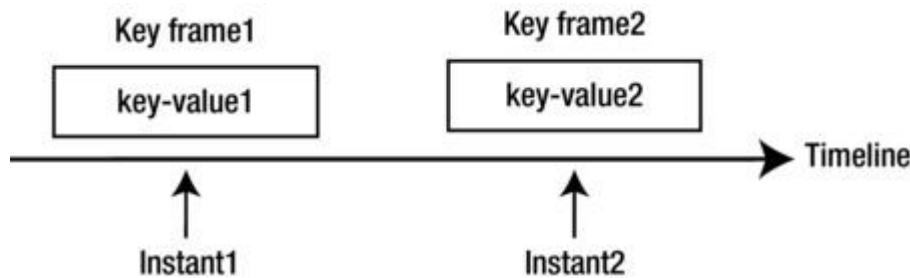
## Understanding the KeyValue Class

An instance of the `KeyValue` class represents a key value that is interpolated for a particular interval during animation. It encapsulates three things.

- A target
- An end value for the target
- An interpolator

The target is a `WritableValue`, which qualifies all JavaFX properties to be a target. The end value is the value for the target at the end of the interval. The interpolator is used to compute the intermediate key frames.

A key frame contains one or more key values and it defines a specific point on a timeline. Figure 22-3 shows an interval on a timeline. The interval is defined by two instants: *instant1* and *instant2*. Both instants have an associated key frame; each key frame contains a key value. An animation may progress forward or backward on the timeline. When an interval starts, the end value of the target is taken from the key value of the end key frame of the interval and its interpolator is used to compute the intermediate key frames. Suppose, in the figure, the animation is progressing in the forward direction and *instant1* occurs before *instant2*. From *instant1* to *instant2*, the interpolator of the `key-value2` will be used to compute the key frames for the interval. If the animation is progressing in the backward direction, the interpolator of the `key-value1` will be used to compute the intermediate key frames from *instant2* to *instant1*.



**Figure 22-3.** Key frames at two instants on a timeline

The `KeyValue` class is immutable. It provides two constructors.

- `KeyValue(WritableValue<T> target, T endValue)`
- `KeyValue(WritableValue<T> target, T endValue, Interpolator interpolator)`

The `Interpolator.LINEAR` is used as the default interpolator that interpolates the animated property linearly with time. I will discuss different types of interpolators later.

The following snippet of code creates a `Text` object and two `KeyValue` objects. The `translateXProperty` is the target. `0` and `100` are the end values for the target. The default interpolator is used.

```
Text msg = new Text("JavaFX animation is cool!");
KeyValue initKeyValue = new KeyValue(msg.translateXProperty(),
0.0);
KeyValue endKeyValue = new KeyValue(msg.translateXProperty(),
100.0);
```

The following snippet of code is similar to the one shown above. It uses the `Interpolator.EASE_BOTH` interpolator, which slows down the animation in the start and toward the end.

```
Text msg = new Text("JavaFX animation is cool!");
KeyValue initKeyValue = new KeyValue(msg.translateXProperty(),
0.0, Interpolator.EASE_BOTH);
KeyValue endKeyValue = new KeyValue(msg.translateXProperty(),
100.0, Interpolator.EASE_BOTH);
```

## Understanding the KeyFrame Class

A key frame defines the target state of a node at a specified point on the timeline. The target state is defined by the key values associated with the key frame.

A key frame encapsulates four things.

- An instant on the timeline
- A set of `KeyValues`
- A name
- An `ActionEvent` handler

The instant on the timeline with which the key frame is associated is defined by a `Duration`, which is an offset of the key frame on the timeline.

- The set of `KeyValues` defines the end value of the target for the key frame.

A key frame may optionally have a name that can be used as a cue point to jump to the instant defined by it during an animation.

The `getCuePoints()` method of the `Animation` class returns a Map of cue points on the Timeline.

Optionally, you can attach an `ActionEvent` handler to a `KeyFrame`. The `ActionEvent` handler is called when the time for the key frame arrives during animation.

An instance of the `KeyFrame` class represents a key frame. The class provides several constructors:

- `KeyFrame(Duration time, EventHandler<ActionEvent> onFinished, KeyValue... values)`
- `KeyFrame(Duration time, KeyValue... values)`
- `KeyFrame(Duration time, String name, EventHandler<ActionEvent> onFinished, Collection<KeyValue> values)`
- `KeyFrame(Duration time, String name, EventHandler<ActionEvent> onFinished, KeyValue... values)`
- `KeyFrame(Duration time, String name, KeyValue... values)`

The following snippet of code creates two instances of `KeyFrame` that specify the `translateX` property of a `Text` node at 0 seconds and 3 seconds on a timeline:

```
Text msg = new Text("JavaFX animation is cool!");
KeyValue initKeyValue = new KeyValue(msg.translateXProperty(), 0.0);
KeyValue endKeyValue = new KeyValue(msg.translateXProperty(), 100.0);

KeyFrame initFrame = new KeyFrame(Duration.ZERO, initKeyValue);
KeyFrame endFrame = new KeyFrame(Duration.seconds(3), endKeyValue);
```

## **Understating the Timeline Animation**

A timeline animation is used for animating any properties of a node. An instance of the `Timeline` class represents a timeline animation. Using a timeline animation involves the following steps:

- Construct key frames
- Create a `Timeline` object with key frames
- Set the animation properties
- Use the `play()` method to run the animation

You can add key frames to a `Timeline` at the time of creating it or after. The `Timeline` instance keeps all key frames in an `ObservableList<KeyFrame>` object.

The `getKeyFrames()` method returns the list. You can modify the list of key frames at any time. If the timeline animation is already running, you need to stop and restart it to pick up the modified list of key frames.

The `Timeline` class contains several constructors.

- `Timeline()`
- `Timeline(double targetFramerate)`
- `Timeline(double targetFramerate, KeyFrame... keyFrames)`
- `Timeline(KeyFrame... keyFrames)`

The no-args constructor creates a `Timeline` with no key frames with animation running at the optimum rate. Other constructors let you specify the target frame rate for the animation, which is the number of frames per second, and the key frames.

Note that the order in which the key frames are added to a `Timeline` is not important. `Timeline` will order them based on their time offset.

The program in Listing 22-2 starts a timeline animation that scrolls a text horizontally from right to left across the scene forever. Figure 22-4 shows a screenshot of the animation.

### ***Listing 22-2.*** Scrolling Text Using a Timeline Animation

```
// ScrollingText.java
package com.jdojo.animation;

import javafx.animation.KeyFrame;
import javafx.animation.KeyValue;
import javafx.animation.Timeline;
import javafx.application.Application;
import javafx.geometry.VPos;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;
import javafx.util.Duration;

public class ScrollingText extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }
}
```

```
@Override
public void start(Stage stage) {
    Text msg = new Text("JavaFX animation is cool!");
    msg.setTextOrigin(VPos.TOP);
    msg.setFont(Font.font(24));

    Pane root = new Pane(msg);
    root.setPrefSize(500, 70);
    Scene scene = new Scene(root);

    stage.setScene(scene);
    stage.setTitle("Scrolling Text");
    stage.show();

    /* Set up a Timeline animation */
    // Get the scene width and the text width
    double sceneWidth = scene.getWidth();
    double msgWidth = msg.getLayoutBounds().getWidth();

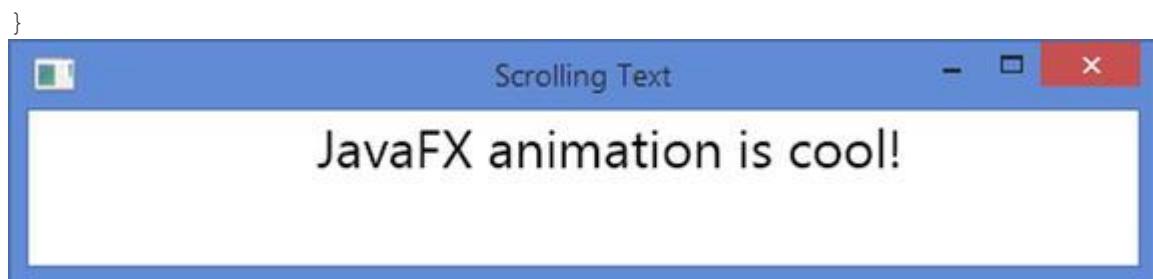
    // Create the initial and final key frames
    KeyValue initKeyValue =
        new KeyValue(msg.translateXProperty(), -
sceneWidth);
    KeyFrame initFrame = new KeyFrame(Duration.ZERO,
initKeyValue);

    KeyValue endKeyValue =
        new KeyValue(msg.translateXProperty(), -
1.0 * msgWidth);
    KeyFrame endFrame = new
KeyFrame(Duration.seconds(3), endKeyValue);

    // Create a Timeline object
    Timeline timeline = new Timeline(initFrame,
endFrame);

    // Let the animation run forever
    timeline.setCycleCount(Timeline.INDEFINITE);

    // Start the animation
    timeline.play();
}
```



**Figure 22-4.** Scrolling text using a timeline animation

The logic to perform the animation is in the `start()` method. The method starts with creating a `Textobject`, a `Pane` with the `Text` object, and setting up a scene for the stage. After showing the stage, it sets up an animation.

It gets the width of the scene and the `Text` object.

```
double sceneWidth = scene.getWidth();
double msgWidth = msg.getLayoutBounds().getWidth();
```

Two key frames are created: one for time = 0 seconds and one for time = 3 seconds. The animation uses the `translateX` property of the `Text` object to change its horizontal position to make it scroll. At 0 seconds, the `Text` is positioned at the scene width, so it is invisible. At 3 seconds, it is placed to the left of the scene at a distance equal to its length, so again it is invisible.

```
KeyValue initKeyValue = new KeyValue(msg.translateXProperty(), sceneWidth);
KeyFrame initFrame = new KeyFrame(Duration.ZERO, initKeyValue);

KeyValue endKeyValue = new KeyValue(msg.translateXProperty(), -1.0 * msgWidth);
KeyFrame endFrame = new KeyFrame(Duration.seconds(3), endKeyValue);
```

A `Timeline` object is created with two key frames.

```
Timeline timeline = new Timeline(initFrame, endFrame);
```

By default, the animation will run only one time. That is, the `Text` will scroll from right to left once and the animation will stop. You can set the cycle count for an animation, which is the number of times the animation needs to run. You run the animation forever by setting the cycle count to `Timeline.INDEFINITE`.

```
timeline.setCycleCount(Timeline.INDEFINITE);
```

Finally, the animation is started by calling the `play()` method.

```
timeline.play();
```

Our example has a flaw. The scrolling the text does not update its initial horizontal position when the width of the scene changes. You can rectify this problem by updating the initial key frame whenever the scene width changes. Append the following statement to the `start()` method of Listing 22-2. It adds a `ChangeListener` for the scene width that updates key frames and restarts the animation.

```
scene.widthProperty().addListener( (prop, oldValue, newValue) ->
{
    KeyValue kv = new KeyValue(msg.translateXProperty(), scene.getWidth());
    KeyFrame kf = new KeyFrame(Duration.ZERO, kv);
    timeline.stop();
    timeline.getKeyFrames().clear();
    timeline.getKeyFrames().addAll(kf, endFrame);
```

```
    timeline.play();
});
```

It is possible to create a `Timeline` animation with only one key frame. The key frame is treated as the last key frame.

The `Timeline` synthesizes an initial key frame (for time = 0 seconds) using the current values for the `WritableValue` being animated. To see the effect, let us replace the statement

```
Timeline timeline = new Timeline(initFrame, endFrame);
```

in Listing 22-2 with the following

```
Timeline timeline = new Timeline(endFrame);
```

The `Timeline` will create an initial key frame with the current value of `translateX` property of the `Text` object, which is 0.0. This time, the `Text` scrolls differently. The scrolling starts by placing the `Text` at 0.0 and scrolling it to the left, so it goes beyond the scene.

## Controlling an Animation

The `Animation` class contains properties and methods that can be used to control animation in various ways. The following sections will explain those properties and methods and how to use them to control animation.

### Playing an Animation

The `Animation` class contains four methods to play an animation.

- `play()`
- `playFrom(Duration time)`
- `playFrom(String cuePoint)`
- `playFromStart()`

The `play()` method plays an animation from its current position. If the animation was never started or stopped, it will play from the beginning. If the animation was paused, it will play from the position where it was paused. You can use the `jumpTo(Duration time)` and `jumpTo(String cuePoint)` methods to set the current position of the animation to a specific duration or a cue point, before calling the `play()` method. Calling the `play()` method is asynchronous. The animation may not start immediately. Calling the `play()` method while animation is running has no effect.

The `playFrom()` method plays an animation from the specified duration or the specified cue point. Calling this method is equivalent to

setting the current position using the `jumpTo()` method and then calling the `play()` method.

The `playFromStart()` method plays the animation from the beginning (`duration = 0`).

## Delaying the Start of an Animation

You can specify a delay in starting the animation using the `delay` property. The value is specified in `Duration`. By default, it is `0 milliseconds`.

```
Timeline timeline = ...
```

```
// Delay the start of the animation by 2 seconds  
timeline.setDelay(Duration.seconds(2));  
  
// Play the animation  
timeline.play();
```

## Stopping an Animation

Use the `stop()` method to stop a running animation. The method has no effect if the animation is not running. The animation may not stop immediately when the method is called as the method executes asynchronously. The method resets the current position to the beginning. That is, calling `play()` after `stop()` will play the animation from the beginning.

```
Timeline timeline = ...  
...  
timeline.play();  
...  
timeline.stop();
```

## Pausing an Animation

Use the `pause()` method to pause an animation. Calling this method when animation is not running has no effect. This method executes asynchronously. Calling the `play()` method when the animation is paused plays it from the current position. If you want to play the animation from the start, call the `playFromStart()` method.

## Knowing the State of an Animation

An animation can be one of the following three states:

- Running
- Paused
- Stopped

The three states are represented by RUNNING, STOPPED, and PAUSED constants of the Animation.Status enum. You do not change the state of an animation directly. It is changed by calling one of the methods of the Animation class. The class contains a read-only status property that can be used to know the state of the animation at any time.

```
Timeline timeline = ...
...
Animation.Status status = timeline.getStatus();
switch(status) {
    case RUNNING:
        System.out.println("Running");
        break;
    case STOPPED:
        System.out.println("Stopped");
        break;
    case PAUSED:
        System.out.println("Paused");
        break;
}
```

## Looping an Animation

An animation can cycle multiple times, even indefinitely. The cycleCount property specifies the number of cycles in an animation, which defaults to 1. If you want to run the animation in an infinite loop, specify Animation.INDEFINITE as the cycleCount. The cycleCount must be set to a value greater than zero. If the cycleCount is changed while the animation is running, the animation must be stopped and restarted to pick up the new value.

```
Timeline timeline1 = ...
Timeline1.setCycleCount(Timeline.INDEFINITE); // Run the
animation forever

Timeline timeline2 = ...
Timeline2.setCycleCount(2); // Run the animation for two cycles
```

## Auto Reversing an Animation

By default, an animation runs only in the forward direction. For example, our scrolling text animation scrolled the text from right to left in one cycle. In the next cycle, the scrolling occurs again from right to left.

Using the autoReverse property, you can define whether the animation is performed in the reverse direction for alternating cycles. By default, it is set to false. Set it to true to reverse the direction of the animation.

```
Timeline timeline = ...
timeline.setAutoReverse(true); // Reverse direction on
alternating cycles
```

If you change the `autoReverse`, you need to stop and restart the animation for the new value to take effect.

### Attaching an `onFinished` Action

You can execute an `ActionEvent` handler when an animation finishes. Stopping the animation or terminating the application while the animation is running will not execute the handler. You can specify the handler in the `onFinished` property of the `Animation` class. The following snippet of code sets the `onFinished` property to an `ActionEvent` handler that prints a message on the standard output:

```
Timeline timeline = ...
timeline.setOnFinished(e -> System.out.print("Animation
finished."));
```

Note that an animation with an `Animation.INDEFINITE` cycle count will not finish and attaching such an action to the animation will never execute.

### Knowing the Duration of an Animation

An animation involves two types of durations.

- Duration to play one cycle of the animation
- Duration to play all cycles of the animation

These durations are not set directly. They are set using other properties of the animation (cycle count, key frames, etc.).

The duration for one cycle is set using key frames. The key frame with the maximum duration determines the duration for one cycle when the animation is played at the rate 1.0. The read-only `cycleDuration` property of the `Animation` class reports the duration for one cycle.

The total duration for an animation is reported by the read-only `totalDuration` property. It is equal to `cycleCount * cycleDuration`. If the `cycleCount` is set to `Animation.INDEFINITE`, the `totalDuration` is reported as `Duration.INDEFINITE`.

Note that the actual duration for an animation depends on its play rate represented by the `rate` property. Because the play rate can be changed while animation is running, there is no easy way to compute the actual duration of an animation.

## Adjusting the Speed of an Animation

The `rate` property of the `Animation` class specifies the direction and the speed for the animation. The sign of its value indicates the direction. The magnitude of the value indicates the speed. A positive value indicates the play in the forward direction. A negative value indicates the play in the backward direction. A value of 1.0 is considered the normal rate of play, a value of 2.0 double the normal rate, 0.50 half the normal rate, and so on. A `rate` of 0.0 stops the play.

It is possible to invert the `rate` of a running animation. In that case, the animation is played in the reverse direction from the current position for the duration that has already elapsed. Note that you cannot start an animation using a negative `rate`. An animation with a negative `rate` will not start. You can change the `rate` to be negative only when the animation has played for a while.

```
Timeline timeline = ...
```

```
// Play the animation at double the normal rate
Timeline.setRate(2.0);
...
timeline.play();
...
// Invert the rate of the play
timeline.setRate(-1.0 * timeline.getRate());
```

The read-only `currentRate` property indicates the current rate (the direction and speed) at which the animation is playing. The values for the `rate` and `currentRate` properties may not be equal.

The `rate` property indicates the rate at which the animation is expected to play when it runs, whereas the `currentRate` indicates the rate at which the animation is being played. When the animation is stopped or paused, the `currentRate` value is 0.0. If the animation reverses its direction automatically, the `currentRate` will report a different direction during reversal; for example, if the `rate` is 1.0, the `currentRate` reports 1.0 for the forward play cycle and -1.0 for the reverse play cycle.

## Understanding Cue Points

You can set up cue points on a timeline. Cue points are named instants on the timeline. An animation can jump to a cue point using the `jumpTo(String cuePoint)` method. An animation maintains an `ObservableMap<String, Duration>` of cue points. The key in the map is the name of the cue points and the values are the corresponding duration on the timeline. Use the `getCuePoints()` method to get the reference of the cue points map.

There are two ways to add cue points to a timeline.

- Giving a name to the `KeyFrame` you add to a timeline that adds a cue point in the cue point map
- Adding name-duration pairs to the map returned by the `getCuePoints()` method of the `Animation` class

**Tip** Every animation has two predefined cue points: “start” and “end.” They are set at the start and end of the animation. The two cue points do not appear in the map returned by the `getCuePoints()` method.

The following snippet of code creates a `KeyFrame` with a name “midway.” When it is added to a timeline, a cue point named “midway” will be added to the timeline automatically. You can jump to this `KeyFrame` using `jumpTo("midway")`.

```
// Create a KeyFrame with name "midway"
KeyValue midKeyValue = ...
KeyFrame midFrame = new KeyFrame(Duration.seconds(5), "midway",
midKeyValue);
```

The following snippet of code adds two cue points directly to the cue point map of a timeline:

```
Timeline timeline = ...
timeline.getCuePoints().put("3 seconds", Duration.seconds(3));
timeline.getCuePoints().put("7 seconds", Duration.seconds(7));
```

The program in Listing 22-3 shows how to add and use cue points on a timeline. It adds a `KeyFrame` with a “midway” name, which automatically becomes a cue point. It adds two cue points, “3 seconds” and “7 seconds,” directly to the cue point map. The list of available cue points is shown in a `ListView` on the left side of the screen. A `Text` object scrolls with a cycle duration of 10 seconds. The program displays a window as shown in Figure 22-5. Select a cue point from the list and the animation will start playing from that point.

### ***Listing 22-3.*** Using Cue Points in Animation

```
// CuePointTest.java
package com.jdojo.animation;

import java.util.Map;
import java.util.SortedMap;
import java.util.TreeMap;
import java.util.Comparator;
import javafx.animation.KeyFrame;
import javafx.animation.KeyValue;
import javafx.animation.Timeline;
import javafx.application.Application;
import javafx.geometry.VPos;
```

```

import javafx.scene.Scene;
import javafx.scene.control.ListView;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.Pane;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;
import javafx.util.Duration;

public class CuePointTest extends Application {
    Text msg = new Text("JavaFX animation is cool!");
    Pane pane;
    ListView<String> cuePointsListView;
    Timeline timeline;

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

    @Override
    public void start(Stage stage) {
        msg.setTextOrigin(VPos.TOP);
        msg.setFont(Font.font(24));

        BorderPane root = new BorderPane();
        root.setPrefSize(600, 150);

        cuePointsListView = new ListView<>();
        cuePointsListView.setPrefSize(100, 150);
        pane = new Pane(msg);

        root.setCenter(pane);
        root.setLeft(cuePointsListView);

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Cue Points");
        stage.show();

        this.setupAnimation();
        this.addCuePoints();
    }

    private void setupAnimation() {
        double paneWidth = pane.getWidth();
        double msgWidth = msg.getLayoutBounds().getWidth();

        // Create the initial and final key frames
        KeyValue initKeyValue = new
        KeyValue(msg.translateXProperty(), paneWidth);
        KeyFrame initFrame = new KeyFrame(Duration.ZERO,
        initKeyValue);

        // A KeyFrame with a name "midway" that defines
        a cue point this name
    }
}

```

```

        KeyValue midKeyValue = new
KeyValue(msg.translateXProperty(), paneWidth / 2);
        KeyFrame midFrame = new
KeyFrame(Duration.seconds(5), "midway", midKeyValue);

        KeyValue endKeyValue = new
KeyValue(msg.translateXProperty(), -1.0 * msgWidth);
        KeyFrame endFrame = new
KeyFrame(Duration.seconds(10), endKeyValue);

        timeline = new Timeline(initFrame, midFrame,
endFrame);
        timeline.setCycleCount(Timeline.INDEFINITE);
        timeline.play();
    }

private void addCuePoints() {
    // Add two cue points directly to the map
    timeline.getCuePoints().put("3 seconds",
Duration.seconds(3));
    timeline.getCuePoints().put("7 seconds",
Duration.seconds(7));

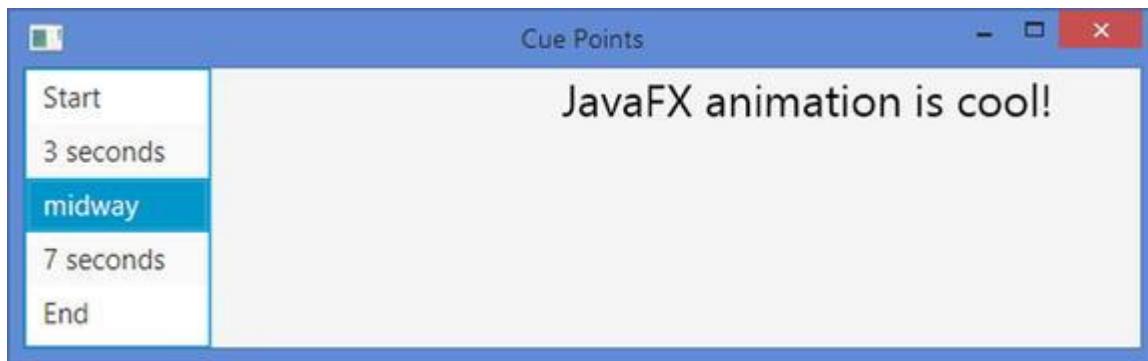
    // Add all cue points from the map to the ListView
    // in the order
    // of their durations
    SortedMap<String, Duration> smap
= getSortedCuePoints(timeline.getCuePoints());
    cuePointsListView.getItems().addAll(smap.keySet());

    // Add the special "start" and "end" cue points
    cuePointsListView.getItems().add(0, "Start");
    cuePointsListView.getItems().add("End");

    // Jusp to the cue point when the user selects it
    cuePointsListView.getSelectionModel().selectedItemProperty()
.addListener(
    (prop, oldValue, newValue) -> {
        timeline.jumpTo(newValue);
    });
}

// Sort the cue points based on their durations
private SortedMap<String, Duration> getSortedCuePoints(
    Map<String, Duration> map) {
    Comparator<String> comparator = (e1, e2) ->
map.get(e1).compareTo(map.get(e2));
    SortedMap<String, Duration> smap = new
TreeMap<>(comparator);
    smap.putAll(map);
    return smap;
}
}

```



**Figure 22-5.** Scrolling text with the list of cue points

## Understanding Transitions

In the previous sections, you saw animations using a timeline that involved setting up key frames on the timeline. Using timeline animation is not easy in all cases. Consider moving a node in a circular path. Creating key frames and setting up a timeline to move the node on the circular path are not easy. JavaFX contains a number of classes (known as *transitions*) that let you animate nodes using predefined properties.

All transition classes inherit from the `Transition` class, which, in turn, inherits from the `Animation` class. All methods and properties in the `Animation` class are also available for use in creating transitions. The transition classes take care of creating the key frames and setting up the timeline. You need to specify the node, duration for the animation, and end values that are interpolated. Special transition classes are available to combine multiple animations that may run sequentially or in parallel.

The `Transition` class contains an `interpolator` property that specifies the interpolator to be used during animation. By default, it uses `Interpolator.EASE_BOTH`, which starts the animation slowly, accelerates it, and slows it down toward the end.

### Understanding the Fade Transition

An instance of the `FadeTransition` class represents a fade-in or fade-out effect for a node by gradually increasing or decreasing the `opacity` of the node over the specified duration. The class defines the following properties to specify the animation:

- `duration`
- `node`
- `fromValue`
- `toValue`
- `byValue`

The `duration` property specifies the duration for one cycle of the animation.

The `node` property specifies the node whose `opacity` property is changed.

The `fromValue` property specifies the initial value for the opacity. If it is not specified, the current `opacity` of the node is used.

The `toValue` property specifies the opacity end value.

The `opacity` of the node is updated between the initial value and the `toValue` for one cycle of the animation.

The `byValue` property lets you specify the `opacity` end value differently using the formula

```
opacity_end_value = opacity_initial_value + byValue
```

The `byValue` lets you set the `opacity` end value by incrementing or decrementing the initial value by an offset. If both `toValue` and `byValue` are specified, the `toValue` is used.

Suppose you want to set the initial and end opacity of a node between 1.0 and 0.5 in an animation. You can achieve it by setting the `fromValue` and `toValue` to 1.0 and 0.50 or by setting `fromValue` and `byValue` to 1.0 and -0.50.

The valid `opacity` value for a node is between 0.0 and 1.0. It is possible to set `FadeTransition` properties to exceed the range. The transition takes care of clamping the actual value in the range.

The following snippet of code sets up a fade-out animation for a `Rectangle` by changing its `opacity` from 1.0 to 0.20 in 2 seconds:

```
Rectangle rect = new Rectangle(200, 50, Color.RED);
FadeTransition fadeInOut = new
FadeTransition(Duration.seconds(2), rect);
fadeInOut.setFromValue(1.0);
fadeInOut.setToValue(.20);
fadeInOut.play();
```

The program in Listing 22-4 creates a fade-out and fade-in effect in an infinite loop for a `Rectangle`.

#### ***Listing 22-4.*** Creating a Fading Effect Using the `FadeTransition` Class

```
// FadeTest.java
package com.jdojo.animation;

import javafx.animation.FadeTransition;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
```

```

import javafx.util.Duration;

public class FadeTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Rectangle rect = new Rectangle(200, 50, Color.RED);
        HBox root = new HBox(rect);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Fade-in and Fade-out");
        stage.show();

        // Set up a fade-in and fade-out animation for the
        rectangle
        FadeTransition fadeInOut = new
        FadeTransition(Duration.seconds(2), rect);
        fadeInOut.setFromValue(1.0);
        fadeInOut.setToValue(.20);
        fadeInOut.setCycleCount(FadeTransition.INDEFINITE);
        fadeInOut.setAutoReverse(true);
        fadeInOut.play();
    }
}

```

## Understanding the Fill Transition

An instance of the `FillTransition` class represents a fill transition for a shape by gradually transitioning the `fill` property of the shape between the specified range and duration. The class defines the following properties to specify the animation:

- `duration`
- `shape`
- `fromValue`
- `toValue`

The `duration` property specifies the duration for one cycle of the animation.

The `shape` property specifies the Shape whose `fill` property is changed.

The `fromValue` property specifies the initial `fill` color. If it is not specified, the current `fill` of the shape is used.

The `toValue` property specifies the `fill` end value.

The `fill` of the shape is updated between the initial value and the `toValue` for one cycle of the animation. The `fill` property in

the `Shape` class is defined as a `Paint`. However, the `fromValue` and `toValue` are of the type `Color`. That is, the fill transition works for two `Colors`, not two `Paints`.

The following snippet of code sets up a fill transition for a `Rectangle` by changing its fill from blue violet to azure in 2 seconds:

```
FillTransition fillTransition = new
FillTransition(Duration.seconds(2), rect);
fillTransition.setFromValue(Color.BLUEVIOLET);
fillTransition.setToValue(Color.AZURE);
fillTransition.play();
```

The program in Listing 22-5 creates a fill transition to change the fill color of a `Rectangle` from blue violet to azure in 2 seconds in an infinite loop.

### ***Listing 22-5.*** Creating a Fill Transition Using the `FillTransition` Class

```
// FillTest.java
package com.jdojo.animation;

import javafx.animation.FillTransition;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.util.Duration;

public class FillTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Rectangle rect = new Rectangle(200, 50, Color.RED);
        HBox root = new HBox(rect);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Fill Transition");
        stage.show();

        // Set up a fill transition for the rectangle
        FillTransition fillTransition = new
        FillTransition(Duration.seconds(2), rect);
        fillTransition.setFromValue(Color.BLUEVIOLET);
        fillTransition.setToValue(Color.AZURE);
        fillTransition.setCycleCount(FillTransition.INDEFINI
TE);
        fillTransition.setAutoReverse(true);
```

```

        fillTransition.play();
    }
}

```

## Understanding the Stroke Transition

An instance of the `StrokeTransition` class represents a stroke transition for a shape by gradually transitioning the `stroke` property of the shape between the specified range and duration. The stroke transition works the same as the fill transition, except that it interpolates the `stroke` property of the shape rather than the `fill` property. The `StrokeTransition` class contains the same properties as the `FillTransition` class. Please refer to the section “Understanding the Fill Transition” for more details. The following snippet of code starts animating the `stroke` of a `Rectangle` in an infinite loop.

The `stroke` changes from red to blue in a cycle duration of 2 seconds.

```

Rectangle rect = new Rectangle(200, 50, Color.WHITE);
StrokeTransition strokeTransition = new
StrokeTransition(Duration.seconds(2), rect);
strokeTransition.setFromValue(Color.RED);
strokeTransition.setToValue(Color.BLUE);
strokeTransition.setCycleCount(StrokeTransition.INDEFINITE);
strokeTransition.setAutoReverse(true);
strokeTransition.play();

```

## Understanding the Translate Transition

An instance of the `TranslateTransition` class represents a translate transition for a node by gradually changing the `translateX`, `translateY`, and `translateZ` properties of the node over the specified duration. The class defines the following properties to specify the animation:

- `duration`
- `node`
- `fromX`
- `fromY`
- `fromZ`
- `toX`
- `toY`
- `toZ`
- `byX`
- `byY`
- `byZ`

The duration property specifies the duration for one cycle of the animation.

The node property specifies the node whose translateX, translateY, and translateZ properties are changed.

The initial location of the node is defined by the (fromX, fromY, fromZ) value. If it is not specified, the current (translateX, translateY, translateZ) value of the node is used as the initial location.

The (toX, toY, toZ) value specifies the end location.

The (byX, byY, byZ) value lets you specify the end location using the following formula:

```
translateX_end_value = translateX_initial_value + byX
translateY_end_value = translateY_initial_value + byY
translateZ_end_value = translateZ_initial_value + byZ
```

If both (toX, toY, toZ) and (byX, byY, byZ) values are specified, the former is used.

The program in Listing 22-6 creates a translate transition in an infinite loop for a Text object by scrolling it across the width of the scene. The program in Listing 22-2 created the same animation using a Timeline object with one difference. They use different interpolators. By default, timeline-based animations use the Interpolator.LINEAR interpolator whereas transition-based animation uses the Interpolator.EASE\_BOTH interpolator. When you run the program in Listing 22-6, the text starts scrolling slow in the beginning and end, whereas in Listing 22-2, the text scrolls with a uniform speed all the time.

### ***Listing 22-6.*** Creating a Translate Transition Using the TranslateTransition Class

```
// TranslateTest.java
package com.jdojo.animation;

import javafx.animation.TranslateTransition;
import javafx.application.Application;
import javafx.geometry.VPos;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;
import javafx.util.Duration;

public class TranslateTest extends Application {
    public static void main(String[] args) {
```

```
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Text msg = new Text("JavaFX animation is cool!");
        msg.setTextOrigin(VPos.TOP);
        msg.setFont(Font.font(24));

        Pane root = new Pane(msg);
        root.setPrefSize(500, 70);
        Scene scene = new Scene(root);

        stage.setScene(scene);
        stage.setTitle("Scrolling Text using a Translate
Transition");
        stage.show();

        // Set up a translate transition for the Text object
        TranslateTransition tt = new
TranslateTransition(Duration.seconds(2), msg);
        tt.setFromX(scene.getWidth());
        tt.setToX(-1.0 * msg.getLayoutBounds().getWidth());
        tt.setCycleCount(TranslateTransition.INDEFINITE);
        tt.setAutoReverse(true);
        tt.play();
    }
}
```

## Understanding the Rotate Transition

An instance of the `RotateTransition` class represents a rotation transition for a node by gradually changing its `rotate` property over the specified duration. The rotation is performed around the center of the node along the specified axis. The class defines the following properties to specify the animation:

- `duration`
- `node`
- `axis`
- `fromAngle`
- `toAngle`
- `byAngle`

The `duration` property specifies the duration for one cycle of the animation.

The `node` property specifies the node whose `rotate` property is changed.

The `axis` property specifies the axis of rotation. If it is unspecified, the value for the `rotationAxis` property, which defaults to `Rotate.Z_AXIS`, for the node is used. The possible values are `Rotate.X_AXIS`, `Rotate.Y_AXIS`, and `Rotate.Z_AXIS`.

The initial angle for the rotation is specified by `fromAngle` property. If it is unspecified, the value for the `rotate` property of the node is used as the initial angle.

The `toAngle` specifies the end rotation angle.

The `byAngle` lets you specify the end rotation angle using the following formula:

```
rotation_end_value = rotation_initial_value + byAngle
```

If both `toAngle` and `byAngle` values are specified, the former is used. All angles are specified in degrees. Zero degrees correspond to the 3 o'clock position. Positive values for angles are measured clockwise.

The program in Listing 22-7 creates a rotate transition in an infinite loop for a Rectangle. It rotates the Rectangle in clockwise and anticlockwise directions in alternate cycles.

### ***Listing 22-7.*** Creating a Rotate Transition Using the `RotateTransition` Class

```
// RotateTest.java
package com.jdojo.animation;

import javafx.animation.RotateTransition;
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.util.Duration;

public class RotateTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Rectangle rect = new Rectangle(50, 50, Color.RED);
        HBox.setMargin(rect, new Insets(20));
        HBox root = new HBox(rect);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Rotate Transition");
        stage.show();
    }
}
```

```

        // Set up a rotate transition the rectangle
        RotateTransition rt = new
RotateTransition(Duration.seconds(2), rect);
        rt.setFromAngle(0.0);
        rt.setToAngle(360.0);
        rt.setCycleCount(RotateTransition.INDEFINITE);
        rt.setAutoReverse(true);
        rt.play();
    }
}

```

## Understanding the Scale Transition

An instance of the `ScaleTransition` class represents a scale transition for a node by gradually changing its `scaleX`, `scaleY`, and `scaleZ` properties over the specified duration. The class defines the following properties to specify the animation:

- `duration`
- `node`
- `fromX`
- `fromY`
- `fromZ`
- `toX`
- `toY`
- `toZ`
- `byX`
- `byY`
- `byZ`

The `duration` property specifies the duration for one cycle of the animation.

The `node` property specifies the node whose `scaleX`, `scaleY`, and `scaleZ` properties are changed.

The initial scale of the node is defined by the (`fromX`, `fromY`, `fromZ`) value. If it is not specified, the current (`scaleX`, `scaleY`, `scaleZ`) value of the node is used as the initial scale.

The (`toX`, `toY`, `toZ`) value specifies the end scale.

The (`byX`, `byY`, `byZ`) value lets you specify the end scale using the following formula:

```

scaleX_end_value = scaleX_initial_value + byX
scaleY_end_value = scaleY_initial_value + byY
scaleZ_end_value = scaleZ_initial_value + byZ

```

If both (`toX`, `toY`, `toZ`) and (`byX`, `byY`, `byZ`) values are specified, the former is used.

The program in Listing 22-8 creates a scale transition in an infinite loop for a Rectangle by changing its width and height between 100% and 20% of their original values in 2 seconds.

### ***Listing 22-8.*** Creating a Scale Transition Using the ScaleTransition Class

```
// ScaleTest.java
package com.jdojo.animation;

import javafx.animation.ScaleTransition;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.util.Duration;

public class ScaleTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Rectangle rect = new Rectangle(200, 50, Color.RED);
        HBox root = new HBox(rect);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Scale Transition");
        stage.show();

        // Set up a scale transition for the rectangle
        ScaleTransition st = new
        ScaleTransition(Duration.seconds(2), rect);
        st.setFromX(1.0);
        st.setToX(0.20);
        st.setFromY(1.0);
        st.setToY(0.20);
        st.setCycleCount(ScaleTransition.INDEFINITE);
        st.setAutoReverse(true);
        st.play();
    }
}
```

### Understanding the Path Transition

An instance of the PathTransition class represents a path transition for a node by gradually changing its translateX and translateY properties to move it along a path

over the specified duration. The path is defined by the outline of a Shape. The class defines the following properties to specify the animation:

- duration
- node
- path
- orientation

The `duration` property specifies the duration for one cycle of the animation.

The `node` property specifies the node whose `rotate` property is changed.

The `path` property defines the path along which the node is moved. It is a Shape. You can use an Arc, a Circle, a Rectangle, an Ellipse, a Path, a SVGPath, and so on as the path.

The moving node may maintain the same upright position or it may be rotated to keep it perpendicular to the tangent of the path at any point along the path. The `orientation` property specifies the upright position of the node along the path. Its value is one of the constants (`NONE` and `ORTHOGONAL_TO_TANGENT`) of the `PathTransition.OrientationType` enum. The default is `NONE`, which maintains the same upright position.

The `ORTHOGONAL_TO_TANGENT` value keeps the node perpendicular to the tangent of the path at any point. Figure 22-6 shows the positions of a Rectangle moving along a Circle using a `PathTransition`. Notice the way the Rectangle is rotated along the path when the `ORTHOGONAL_TO_TANGENT` orientation is used.



**Figure 22-6.** Effect of using the orientation property of the `PathTransition` class

You can specify the duration, path, and node for the path transition using the properties of the `PathTransition` class or in the constructors. The class contains the following constructors:

- `PathTransition()`
- `PathTransition(Duration duration, Shape path)`

- PathTransition(Duration duration, Shape path, Node node)

The program in Listing 22-9 creates a path transition in an infinite loop for a Rectangle. It moves the Rectangle along a circular path defined by the outline of a Circle.

### ***Listing 22-9.*** Creating a Path Transition Using the PathTransition Class

```
// PathTest.java
package com.jdojo.animation;

import javafx.animation.PathTransition;
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.util.Duration;

public class PathTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // Create the node
        Rectangle rect = new Rectangle(20, 10, Color.RED);

        // Create the path
        Circle path = new Circle(100, 100, 100);
        path.setFill(null);
        path.setStroke(Color.BLACK);

        Group root = new Group(rect, path);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Path Transition");
        stage.show();

        // Set up a path transition for the rectangle
        PathTransition pt = new
PathTransition(Duration.seconds(2), path, rect);
        pt.setOrientation(PathTransition.OrientationType.ORTHOGONAL_TO_TANGENT);
        pt.setCycleCount(PathTransition.INDEFINITE);
        pt.setAutoReverse(true);
        pt.play();
    }
}
```

```
}
```

## Understanding the Pause Transition

An instance of the `PauseTransition` class represents a pause transition. It causes a delay of the specified duration. Its use is not obvious. It is not used alone. Typically, it is used in a sequential transition to insert a pause between two transitions. It defines a `duration` property to specify the duration of the delay.

A pause transition is also useful if you want to execute an `ActionEvent` handler after a specified duration when a transition is finished. You can achieve this by setting its `onFinished` property, which is defined in the `Animation` class.

```
// Create a pause transition of 400 milliseconds that is the
default duration
PauseTransition pt1 = new PauseTransition();

// Change the duration to 10 seconds
pt1.setDuration(Duration.seconds(10));

// Create a pause transition of 5 seconds
PauseTransition pt2 = new PauseTransition(Duration.seconds(5));
```

If you change the duration of a running pause transition, you need to stop and restart the transition to pick up the new duration. You will have an example when I discuss the sequential transition.

## Understanding the Sequential Transition

An instance of the `SequentialTransition` class represents a sequential transition. It executes a list of animations in sequential order. The list of animation may contain timeline-based animations, transition-based animations, or both.

The `SequentialTransition` class contains a `node` property that is used as the node for animations in the list if the animation does not specify a node. If all animations specify a node, this property is not used.

A `SequentialTransition` maintains the animations in an `ObservableList<Animation>`. The `getChildren()` method returns the reference of the list.

The following snippet of code creates a fade transition, a pause transition, and a path transition. Three transitions are added to a sequential transition. When the sequential transition is played, it will play the fade transition, pause transition, and the path transition in sequence.

```
FadeTransition fadeTransition = ...
PauseTransition pauseTransition = ...
```

```

PathTransition pathTransition = ...

SequentialTransition st = new SequentialTransition();
st.getChildren().addAll(fadeTransition, pauseTransition,
pathTransition);
st.play();

```

**Tip** The SequentialTransition class contains constructors that let you specify the list of animations and node.

The program in Listing 22-10 creates a scale transition, a fill transition, a pause transition, and a path transition, which are added to a sequential transition. The sequential transition runs in an infinite loop. When the program runs

- It scales up the rectangle to double its size, and then down to the original size.
- It changes the fill color of the rectangle from red to blue, and then, back to red.
- It pauses for 200 milliseconds, and then, prints a message on the standard output.
- It moves the rectangle along the outline of a circle.
- The foregoing sequence of animations is repeated indefinitely.

### ***Listing 22-10.*** Creating a Sequential Transition Using the SequentialTransition Class

```

// SequentialTest.java
package com.jdojo.animation;

import javafx.animation.FillTransition;
import javafx.animation.PathTransition;
import javafx.animation.PauseTransition;
import javafx.animation.ScaleTransition;
import javafx.animation.SequentialTransition;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.util.Duration;
import static
javafx.animation.PathTransition.OrientationType.ORTHOGONAL_TO_TAN
GENT;

public class SequentialTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }
}

```

```

}

@Override
public void start(Stage stage) {
    // Create the node to be animated
    Rectangle rect = new Rectangle(20, 10, Color.RED);

    // Create the path
    Circle path = new Circle(100, 100, 75);
    path.setFill(null);
    path.setStroke(Color.BLACK);

    Pane root = new Pane(rect, path);
    root.setPrefSize(200, 200);
    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("Sequential Transition");
    stage.show();

    // Set up a scale transition
    ScaleTransition scaleTransition = new
ScaleTransition(Duration.seconds(1));
    scaleTransition.setFromX(1.0);
    scaleTransition.setToX(2.0);
    scaleTransition.setFromY(1.0);
    scaleTransition.setToY(2.0);
    scaleTransition.setCycleCount(2);
    scaleTransition.setAutoReverse(true);

    // Set up a fill transition
    FillTransition fillTransition = new
FillTransition(Duration.seconds(1));
    fillTransition.setFromValue(Color.RED);
    fillTransition.setToValue(Color.BLUE);
    fillTransition.setCycleCount(2);
    fillTransition.setAutoReverse(true);

    // Set up a pause transition
    PauseTransition pauseTransition = new
PauseTransition(Duration.millis(200));
    pauseTransition.setOnFinished(e ->
System.out.println("Ready to circle..."));

    // Set up a path transition
    PathTransition pathTransition = new
PathTransition(Duration.seconds(2), path);
    pathTransition.setOrientation(ORTHOGONAL_TO_TANGENT)
;

    // Create a sequential transition
    SequentialTransition st = new
SequentialTransition();

    // Rectangle is the node for all animations
    st.setNode(rect);
}

```

```

        // Add animations to the list
        st.getChildren().addAll(scaleTransition,
                               fillTransition,
                               pauseTransition,
                               pathTransition);
        st.setCycleCount(PathTransition.INDEFINITE);
        st.play();
    }
}

```

## Understanding the Parallel Transition

An instance of the `ParallelTransition` class represents a parallel transition. It executes a list of animations simultaneously. The list of animations may contain timeline-based animations, transition-based animations, or both.

The `ParallelTransition` class contains a `node` property that is used as the node for animations in the list if the animation does not specify a node. If all animations specify a node, this property is not used.

A `ParallelTransition` maintains the animations in an `ObservableList<Animation>`. The `getChildren()` method returns the reference of the list.

The following snippet of code creates a fade transition and a path transition. They transitions are added to a parallel transition. When the sequential transition is played, it will apply the fading effect and move the node at the same time.

```

FadeTransition fadeTransition = ...
PathTransition pathTransition = ...

ParallelTransition pt = new ParallelTransition();
pt.getChildren().addAll(fadeTransition, pathTransition);
pt.play();

```

**Tip** The `ParallelTransition` class contains constructors that let you specify the list of animations and node.

The program in Listing 22-11 creates a fade transition and a rotate transition. It adds them to a parallel transition. When the program is run, the rectangle rotates and fades in/out at the same time.

### ***Listing 22-11.*** Creating a Parallel Transition Using the `ParallelTransition` Class

```

// ParallelTest.java
package com.jdojo.animation;

import javafx.animation.FadeTransition;
import javafx.animation.ParallelTransition;

```

```
import javafx.animation.PathTransition;
import javafx.animation.RotateTransition;
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.util.Duration;

public class ParallelTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        Rectangle rect = new Rectangle(100, 100, Color.RED);
        HBox.setMargin(rect, new Insets(20));

        HBox root = new HBox(rect);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Parallel Transition");
        stage.show();

        // Set up a fade transition
        FadeTransition fadeTransition = new
FadeTransition(Duration.seconds(1));
        fadeTransition.setFromValue(0.20);
        fadeTransition.setToValue(1.0);
        fadeTransition.setCycleCount(2);
        fadeTransition.setAutoReverse(true);

        // Set up a rotate transition
        RotateTransition rotateTransition =
            new RotateTransition(Duration.seconds(2));
        rotateTransition.setFromAngle(0.0);
        rotateTransition.setToAngle(360.0);
        rotateTransition.setCycleCount(2);
        rotateTransition.setAutoReverse(true);

        // Create and start a sequential transition
        ParallelTransition pt = new ParallelTransition();

        // Rectangle is the node for all animations
        pt.setNode(rect);
        pt.getChildren().addAll(fadeTransition,
rotateTransition);
        pt.setCycleCount(PathTransition.INDEFINITE);
        pt.play();
    }
}
```

## Understanding Interpolators

An interpolator is an instance of the abstract `Interpolator` class. An interpolator plays an important role in an animation. Its job is to compute the key values for the intermediate key frames during animation. Implementing a custom interpolator is easy. You need to subclass the `Interpolator` class and override its `curve()` method. The `curve()` method is passed the time elapsed for the current interval. The time is normalized between 0.0 and 1.0. The start and end of the interval have the value of 0.0 and 1.0, respectively. The value passed to the method would be 0.50 when half of the interval time has elapsed. The return value of the method indicates the fraction of change in the animated property.

The following interpolator is known as a linear interpolator whose `curve()` method returns the passed in argument value:

```
Interpolator linearInterpolator = new Interpolator() {
    @Override
    protected double curve(double timeFraction) {
        return timeFraction;
    }
};
```

The linear interpolator mandates that the percentage of change in the animated property is the same as the progression of the time for the interval.

Once you have a custom interpolator, you can use it in constructing key values for key frames in a timeline-based animation. For a transition-based animation, you can use it as the `interpolator` property of the transition classes.

The animation API calls the `interpolate()` method of the `Interpolator`. If the animated property is an instance of `Number`, it returns

```
startValue + (endValue - startValue) * curve(timeFraction)
```

Otherwise, if the animated property is an instance of the `Interpolatable`, it delegates the interpolation work to the `interpolate()` method of the `Interpolatable`. Otherwise, the interpolator defaults to a discrete interpolator by returning 1.0 when the time fraction is 1.0, and 0.0 otherwise.

JavaFX provides some standard interpolators that are commonly used in animations. They are available as constants in the `Interpolator` class or as its static methods.

- Linear interpolator
- Discrete interpolator
- Ease-in interpolator

- Ease-out interpolator
- Ease-both interpolator
- Spline interpolator
- Tangent interpolator

## Understanding the Linear Interpolator

The `Interpolator.LINEAR` constant represents a linear interpolator. It interpolates the value of the animated property of a node linearly with time. The percentage change in the property for an interval is the same as the percentage of the time passed.

## Understanding the Discrete Interpolator

The `Interpolator.DISCRETE` constant represents a discrete interpolator. A discrete interpolator jumps from one key frame to the next, providing no intermediate key frame. The `curve()` method of the interpolator returns 1.0 when the time fraction is 1.0, and 0.0 otherwise. That is, the animated property value stays at its initial value for the entire duration of the interval. It jumps to the end value at the end of the interval. The program in Listing 22-12 uses discrete interpolators for all key frames. When you run the program, it moves text jumping from key frame to another. Compare this example with the scrolling text example, which used a linear interpolator. The scrolling text example moved the text smoothly whereas this example created a jerk in the movement.

### ***Listing 22-12.*** Using a Discrete Interpolator to Animate Hopping Text

```
// HoppingText.java
package com.jdojo.animation;

import javafx.animation.Interpolator;
import javafx.animation.KeyFrame;
import javafx.animation.KeyValue;
import javafx.animation.Timeline;
import javafx.application.Application;
import javafx.geometry.VPos;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;
import javafx.util.Duration;

public class HoppingText extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }
}
```

```

@Override
public void start(Stage stage) {
    Text msg = new Text("Hopping text!");
    msg.setTextOrigin(VPos.TOP);
    msg.setFont(Font.font(24));

    Pane root = new Pane(msg);
    root.setPrefSize(500, 70);
    Scene scene = new Scene(root);

    stage.setScene(scene);
    stage.setTitle("Hopping Text");
    stage.show();

    // Setup a Timeline animation
    double start = scene.getWidth();
    double end = -1.0
    * msg.getLayoutBounds().getWidth();

    KeyFrame[] frame = new KeyFrame[11];
    for(int i = 0; i <= 10; i++) {
        double pos = start - (start - end) * i / 10.0;

        // Set 2.0 seconds as the cycle duration
        double duration = i/5.0;

        // Use a discrete interpolator
        KeyValue keyValue = new
        KeyValue(msg.translateXProperty(),
                pos,
                Interpolator.DISCRETE
E);
        frame[i] = new
        KeyFrame(Duration.seconds(duration), keyValue);
    }

    Timeline timeline = new Timeline();
    timeline.getKeyFrames().addAll(frame);
    timeline.setCycleCount(Timeline.INDEFINITE);
    timeline.setAutoReverse(true);
    timeline.play();
}
}

```

## Understanding the Ease-In Interpolator

The `Interpolator.EASE_IN` constant represents an ease-in interpolator. It starts the animation slowly for the first 20% of the time interval and accelerates afterward.

## Understanding the Ease-Out Interpolator

The `Interpolator.EASE_OUT` constant represents an ease-out interpolator. It plays animation at a constant speed up to 80% of the time interval and slows down afterwards.

### Understanding the Ease-Both Interpolator

The `Interpolator.EASE_BOTH` constant represents an ease-both interpolator. It plays the animation slower in the first 20% and the last 20% of the time interval and maintains a constant speed otherwise.

### Understanding the Spline Interpolator

The `Interpolator.SPLINE(double x1, double y1, double x2, double y2)` static method returns a spline interpolator. It uses a cubic spline shape to compute the speed of the animation at any point in the interval. The parameters  $(x_1, y_1)$  and  $(x_2, y_2)$  define the control points of the cubic spline shape with  $(0, 0)$  and  $(1, 1)$  as implicit anchor points. The values of the parameters are between 0.0 and 1.0.

The slope at a given point on the cubic spline shape defines the acceleration at that point. A slope approaching the horizontal line indicates deceleration whereas a slope approaching the vertical line indicates acceleration. For example, using  $(0, 0, 1, 1)$  as the parameters to the `SPLINE` method creates an interpolator with a constant speed whereas the parameters  $(0.5, 0, 0.5, 1.0)$  will create an interpolator that accelerates in the first half and decelerates in the second half. Please refer to <http://www.w3.org/TR/SMIL/smil-animation.html#animationNS-OverviewSpline> for more details.

### Understanding the Tangent Interpolator

The `Interpolator.TANGENT` static method returns a tangent interpolator, which defines the behavior of an animation before and after a key frame. All other interpolators interpolate data between two key frames. If you specify a tangent interpolator for a key frame, it is used to interpolate data before and after the key frame. The animation curve is defined in terms of a tangent, which is known as in-tangent, at a specified duration before the key frame and a tangent, which is called an out-tangent, at a specified duration after the key frame. This interpolator is used only in timeline-based animations as it affects two intervals.

The `TANGENT` static method is overloaded.

- `Interpolator TANGENT(Duration t1, double v1, Duration t2, double v2)`
- `Interpolator TANGENT(Duration t, double v)`

In the first version, the parameters  $t_1$  and  $t_2$  are the duration before and after the key frame, respectively. The parameters  $v_1$  and  $v_2$  are the in-tangent and out-tangent values. That is,  $v_1$  is the tangent value at duration  $t_1$  and  $v_2$  is the tangent value at duration  $t_2$ . The second version specifies the same value for both pairs.

## Summary

In JavaFX, animation is defined as changing the property of a node over time. If the property that changes determines the location of the node, the animation in JavaFX will produce an illusion of motion. Not all animations have to involve motion; for example, changing the `fill` property of a `Shape` over time is an animation in JavaFX that does not involve motion.

Animation is performed over a period of time. A *timeline* denotes the progression of time during animation with an associated key frame at a given instant. A *key frame* represents the state of the node being animated at a specific instant on the timeline. A key frame has associated key values. A *key value* represents the value of a property of the node along with an interpolator to be used.

A timeline animation is used for animating any properties of a node. An instance of the `Timeline` class represents a timeline animation.

Using a timeline animation involves the following steps: constructing key frames, creating a `Timeline` object with key frames, setting the animation properties, and using the `play()` method to run the animation. You can add key frames to a `Timeline` at the time of creating it or after. The `Timeline` instance keeps all key frames in an `ObservableList<KeyFrame>` object.

The `getKeyFrames()` method returns the list. You can modify the list of key frames at any time. If the timeline animation is already running, you need to stop and restart it to pick up the modified list of key frames.

The `Animation` class contains several properties and methods to control animation such as playing, reversing, pausing, and stopping.

You can set up cue points on a timeline. Cue points are named instants on the timeline. An animation can jump to a cue point using the `jumpTo(String cuePoint)` method.

Using timeline animation is not easy in all cases. JavaFX contains a number of classes (known as *transitions*) that let you animate nodes using predefined properties. All transition classes inherit from the `Transition` class, which, in turn, inherits from the `Animation` class. The transition classes take care of creating the key frames and setting up the timeline. You need to specify the node,

duration for the animation, and end values that are interpolated. Special transition classes are available to combine multiple animations that may run sequentially or in parallel. The `Transition` class contains an `interpolator` property that specifies the interpolator to be used during animation. By default, it uses `Interpolator.EASE_BOTH`, which starts the animation slowly, accelerates it, and slows it down toward the end.

An interpolator is an instance of the abstract `Interpolator` class. Its job is to compute the key values for the intermediate key frames during animation. JavaFX provides several built-in interpolators such as linear, discrete, ease-in, and ease-out. You can also implement a custom interpolator easily. You need to subclass the `Interpolator` class and override its `curve()` method. The `curve()` method is passed the time elapsed for the current interval. The time is normalized between 0.0 and 1.0. The return value of the method indicates the fraction of change in the animated property.

The next chapter will discuss how to incorporate different types of charts in a JavaFX application.

## CHAPTER 24



### Understanding the Image API

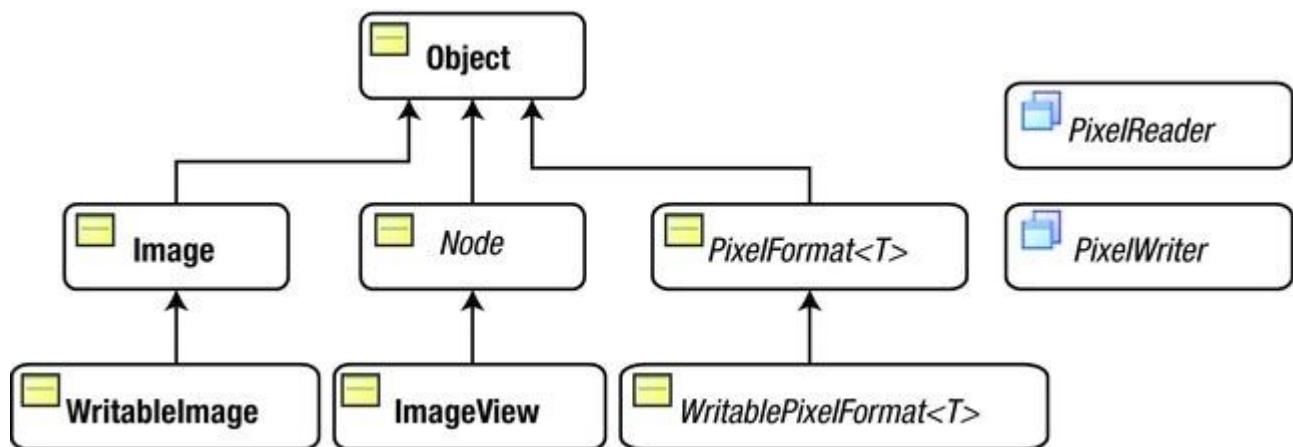
In this chapter, you will learn:

- What the Image API is
- How to load an image
- How to view an image in an `ImageView` node
- How to perform image operations such as reading/writing pixels, creating an image from scratch, and saving the image to the file system
- How to take the snapshot of nodes and scenes

### What Is the Image API?

JavaFX provides the Image API that lets you load and display images, and read/write raw image pixels. A class diagram for the classes in the image API is shown in Figure 24-1. All classes are in the `javafx.scene.image` package. The API lets you

- Load an image in memory
- Display an image as a node in a scene graph
- Read pixels from an image
- Write pixels to an image
- Convert a node in a scene graph to an image and save it to the local file system



**Figure 24-1.** A class diagram for classes in the image API

An instance of the `Image` class represents an image in memory. You can construct an image in a JavaFX application by supplying pixels to a `WritableImage` instance.

An `ImageView` is a `Node`. It is used to display an `Image` in a scene graph. If you want to display an image in an application, you need to load the image in an `Image` and display the `Image` in an `ImageView`.

Images are constructed from pixels. Data for pixels in an image may be stored in different formats. A `PixelFormat` defines how the data for a pixel for a given format is stored.

A `WritablePixelFormat` represents a destination format to write pixels with full pixel color information.

The `PixelReader` and `PixelWriter` interfaces define methods to read from an `Image` and write data to a `WritableImage`. Besides an `Image`, you can read pixels from and write pixels to any surface that contain pixels.

I will cover examples of using these classes in the sections to follow.

## Loading an Image

An instance of the `Image` class is an in-memory representation of an image. The class supports BMP, PNG, JPEG, and GIF image formats. It loads an image from a source, which can be specified as a string URL or an `InputStream`. It can also scale the original image while loading.

The `Image` class contains several constructors that let you specify the properties for the loaded image:

- `Image(InputStream is)`
- `Image(InputStream is, double requestedWidth, double requestedHeight, boolean preserveRatio, boolean smooth)`
- `Image(String url)`
- `Image(String url, boolean backgroundLoading)`
- `Image(String url, double requestedWidth, double requestedHeight, boolean preserveRatio, boolean smooth)`
- `Image(String url, double requestedWidth, double requestedHeight, boolean preserveRatio, boolean smooth, boolean backgroundLoading)`

There is no ambiguity of the source of the image if an `InputStream` is specified as the source. If a string URL is specified as the source, it could be a valid URL or a valid path in the

**CLASSPATH.** If the specified URL is not a valid URL, it is used as a path and the image source will be searched on the path in the CLASSPATH.

```
// Load an image from local machine using an InputStream
String sourcePath = "C:\\mypicture.png";
Image img = new Image(new FileInputStream(sourcePath));

// Load an image from a URL
Image img = new Image("http://jdojo.com/wp-
content/uploads/2013/03/randomness.jpg");

// Load an image from the CLASSPATH. The image is located in the
resources.picture package
Image img = new Image("resources/picture/randomness.jpg");
```

In the above statement, the specified URL resources/picture/randomness.jpg is not a valid URL. The Image class will treat it as a path expecting it to exist in the CLASSPATH. It treats the resource.picture as a package and the randomness.jpg as a resource in that package.

### Specifying the Image-Loading Properties

Some constructors let you specify some image-loading properties to controls the quality of the image and the loading process:

- requestedWidth
- requestedHeight
- preserveRatio
- smooth
- backgroundLoading

The requestedWidth and requestedHeight properties specify the scaled width and height of the image. By default, an image is loaded in its original size.

The preserveRatio property specifies whether to preserve the aspect ratio of the image while scaling. By default, it is false.

The smooth property specifies the quality of the filtering algorithm to be used in scaling. By default, it is false. If it is set to true, a better quality filtering algorithm is used, which slows down the image-loading process a bit.

The backgroundLoading property specifies whether to load the image asynchronously. By default, the property is set to false and the image is loaded synchronously. The loading process starts when the Image object is created. If this property is set to true, the image is loaded asynchronously in a background thread.

### Reading the Loaded-Image Properties

The `Image` class contains the following read-only properties:

- `width`
- `height`
- `progress`
- `error`
- `exception`

The `width` and `height` properties are the width and height of the loaded image, respectively. They are zero if the image failed to load.

The `progress` property indicates the progress in loading the image data. It is useful to know the progress when the `backgroundLoading` property is set to true. Its value is between 0.0 and 1.0 where 0.0 indicates zero percent loading and 1.0 indicates hundred percent loading. When the `backgroundLoading` property is set to false (the default), its value is 1.0. You can add a `ChangeListener` to the `progress` property to know the progress in image loading. You may display a text as a placeholder for an image while it is loading and update the text with the current progress in the `ChangeListener`.

```
// Load an image in the background
String imagePath = "resources/picture/randomness.jpg";
Boolean backgroundLoading = true;
Image image = new Image(imagePath, backgroundLoading);

// Print the loading progress on the standard output
image.progressProperty().addListener((prop, oldValue, newValue) -> {
    System.out.println("Loading:" +
+ Math.round(newValue.doubleValue() * 100.0) + "%");
});
```

The `error` property indicates whether an error occurred while loading the image. If it is true, the `exception` property specifies the `Exception` that caused the error. At the time of this writing, TIFF image format is not supported on Windows. The following snippet of code attempts to load a TIFF image on Windows XP and it produces an error. The code contains an error handling logic that adds a `ChangeListener` to the `error` property if `backgroundLoading` is true. Otherwise, it checks for the value of the `error` property.

```
String imagePath = "resources/picture/test.tif";
Boolean backgroundLoading = false;
Image image = new Image(imagePath, backgroundLoading);

// Add a ChangeListener to the error property for background
loading and
// check its value for non-backgroudn loading
```

```

if (image.isBackgroundLoading()) {
    image.errorProperty().addListener((prop, oldValue,
newValue) -> {
        if (newValue) {
            System.out.println("An error occurred while
loading the image.\n" +
                    "Error message: "
+ image.getException().getMessage());
        }
    });
}
else if (image.isError()) {
    System.out.println("An error occurred while loading the
image.\n" +
                    "Error message: "
+ image.getException().getMessage());
}
An error occurred while loading the image.
Error message: No loader for image data

```

## Viewing an Image

An instance of the `ImageView` class is used to display an image loaded in an `Image` object. The `ImageView` class inherits from the `Node` class, which makes an `ImageView` suitable to be added to a scene graph. The class contains several constructors:

- `ImageView()`
- `ImageView(Image image)`
- `ImageView(String url)`

The no-args constructor creates an `ImageView` without an image. Use the `image` property to set an image. The second constructor accepts the reference of an `Image`. The third constructor lets you specify the URL of the image source. Internally, it creates an `Image` using the specified URL.

```

// Create an empty ImageView and set an Image for it later
ImageView imageView = new ImageView();
imageView.setImage(new
Image("resources/picture/randomness.jpg"));

// Create an ImageView with an Image
ImageView imageView = new ImageView(new
Image("resources/picture/randomness.jpg"));

// Create an ImageView with the URL of the image source
ImageView imageView = new
ImageView("resources/picture/randomness.jpg");

```

The program in Listing 24-1 shows how to display an image in a scene. It loads an image in an `Imageobject`. The image is scaled without preserving the aspect ratio. The `Image` object is added to an `ImageView`, which is added to an `HBox`. Figure 24-2 shows the window.

### ***Listing 24-1.*** Displaying an Image in an ImageView Node

```
// ImageTest.java
package com.jdojo.image;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class ImageTest extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        String imagePath
= "resources/picture/randomness.jpg";

        // Scale the iamge to 200 x 100
        double requestedWidth = 200;
        double requestedHeight = 100;
        boolean preserveRatio = false;
        boolean smooth = true;
        Image image = new Image(imagePath,
                               requestedWidth,
                               requestedHeight,
                               preserveRatio,
                               smooth);
        ImageView imageView = new ImageView(image);

        HBox root = new HBox(imageView);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Displaying an Image");
        stage.show();
    }
}
```



**Figure 24-2.** A window with an image

### Multiple Views of an Image

An `Image` loads an image in memory from its source. You can have multiple views of the same `Image`. An `ImageView` provides one of the views.

You have an option to resize the original image while loading, displaying, or at both times. Which option you choose to resize an image depends on the requirement at hand.

- Resizing an image in an `Image` object resizes the image permanently in memory and all views of the image will use the resized image. Once an `Image` is resized, its size cannot be altered. You may want to reduce the size of an image in an `Image` object to save memory.
- Resizing an image in an `ImageView` resizes the image only for this view. You can resize the view of an image in an `ImageView` even after the image has been displayed.

We have already discussed how to resize an image in an `Image` object. In this section, we will discuss resizing an image in an `ImageView`.

Similar to the `Image` class, the `ImageView` class contains the following four properties to control the resizing of view of an image.

- `fitWidth`
- `fitHeight`
- `preserveRatio`
- `smooth`

The `fitWidth` and `fitHeight` properties specify the resized width and height of the image, respectively. By default, they are zero, which

means that the `ImageView` will use the width and height of the loaded image in the `Image`.

The `preserveRatio` property specifies whether to preserve the aspect ratio of the image while resizing. By default, it is false.

The `smooth` property specifies the quality of the filtering algorithm to be used in resizing. Its default value is platform dependent. If it is set to true, a better quality filtering algorithm is used.

The program in Listing 24-2 loads an image in an `Image` object in original size. It creates three `ImageView` objects of the `Image` specifying different sizes. Figure 24-3 shows the three images. The image shows a junk school bus and a junk car. The image is used with a permission from Richard Castillo (<http://www.digitizedchaos.com>).

### ***Listing 24-2.*** Displaying the Same Image in Different ImageView in Different Sizes

```
// MultipleImageViews.java
package com.jdojo.image;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class MultipleImageViews extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // Load an image in its original size
        String imagePath
= "resources/picture/school_bus.jpg";
        Image image = new Image(imagePath);

        // Create three views of different sizes of the same
image
        ImageView view1 = getImageView(image, 100, 50,
false);
        ImageView view2 = getImageView(image, 100, 50, true);
        ImageView view3 = getImageView(image, 100, 100,
true);

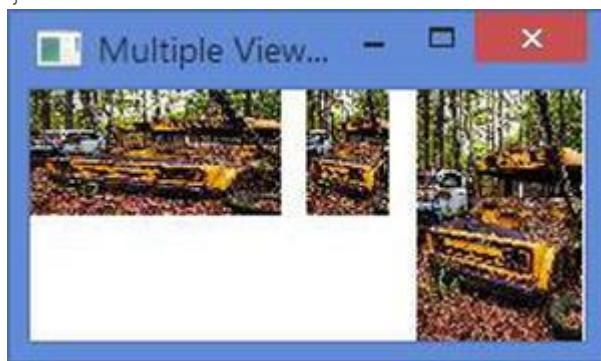
        HBox root = new HBox(10, view1, view2, view3);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Multiple Views of an Image");
    }
}
```

```

        stage.show();
    }

    private ImageView getImageView(Image image,
                                   double fitWidth,
                                   double fitHeight,
                                   boolean preserveRatio) {
        ImageView view = new ImageView(image);
        view.setFitWidth(fitWidth);
        view.setFitHeight(fitHeight);
        view.setPreserveRatio(preserveRatio);
        view.setSmooth(true);
        return view;
    }
}

```



**Figure 24-3.** Three views of the same image

### Viewing an Image in a Viewport

A viewport is a rectangular region to view part of a graphics. It is common to use scrollbars in conjunction with a viewport. As the scrollbars are scrolled, the viewport shows different part of the graphics.

An `ImageView` lets you define a viewport for an image. In JavaFX, a viewport is an instance of the `javafx.geometry.Rectangle2D` object. A `Rectangle2D` is immutable. It is defined in terms of four properties: `minX`, `minY`, `width`, and `height`. The (`minX`, `minY`) value defines the location of the upper-left corner of the rectangle. The `width` and `height` properties specify its size. You must specify all properties in the constructor.

```
// Create a viewport located at (0, 0) and of size 200 x 100
Rectangle2D viewport = new Rectangle2D(0, 0, 200, 100);
```

The `ImageView` class contains a `viewport` property, which provides a viewport into the image displayed in the `ImageView`.

The `viewport` defines a rectangular region in the image.

The `ImageView` shows only the region of the image that falls inside the viewport. The location of the viewport is defined relative to the image,

not the `ImageView`. By default, the viewport of an `ImageView` is null and the `ImageView` shows the whole image.

The following snippet of code loads an image in its original size in an `Image`. The `Image` is set as the source for an `ImageView`. A viewport 200 X 100 in size is set for the `ImageView`. The viewport is located at (0, 0). This shows in the `ImageView` the top-left 200 X 100 region of the image

```
String imagePath = "resources/picture/school_bus.jpg";
Image image = new Image(imagePath);
imageView = new ImageView(image);
Rectangle2D viewport = new Rectangle2D(0, 0, 200, 100);
imageView.setViewport(viewport);
```

The following snippet of code will change the view port to show the 200 X 100 lower-right region of the image.

```
double minX = image.getWidth() - 200;
double minY = image.getHeight() - 100;
Rectangle2D viewport2 = new Rectangle2D(minX, minY, 200, 100);
imageView.setViewport(viewport2);
```

**Tip** The `Rectangle2D` class is immutable. Therefore, you need to create a new viewport every time you want to move the viewport into the image.

The program in Listing 24-3 loads an image into an `ImageView`. It sets a viewport for the `ImageView`. You can drag the mouse, while pressing the left, right, or both buttons, to scroll to the different parts of the image into the view.

### ***Listing 24-3.*** Using a Viewport to View Part of an Image

```
// ImageViewPort.java
package com.jdojo.image;

import javafx.application.Application;
import javafx.geometry.Rectangle2D;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class ImageViewPort extends Application {
    private static final double VIEWPORT_WIDTH = 300;
    private static final double VIEWPORT_HEIGHT = 200;
    private double startX;
    private double startY;
    private ImageView imageView;

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

```

    }

@Override
public void start(Stage stage) {
    // Load an image in its original size
    String imagePath
= "resources/picture/school_bus.jpg";
    Image image = new Image(imagePath);
    imageView = new ImageView(image);

    // Set a viewport for the ImageView
    Rectangle2D viewport = new Rectangle2D(0, 0,
VIEWPORT_WIDTH, VIEWPORT_HEIGHT);
    imageView.setViewport(viewport);

    // Set the mouse pressed and mouse dragged event
hanlders
    imageView.setOnMousePressed(this::handleMousePressed)
;
    imageView.setOnMouseDragged(this::handleMouseDragged)
;

    HBox root = new HBox(imageView);
    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("Viewing an Image in a Viewport");
    stage.show();
}

private void handleMousePressed(MouseEvent e) {
    startX = e.getX();
    startY = e.getY();
}

private void handleMouseDragged(MouseEvent e) {
    // How far the mouse was dragged
    double draggedDistanceX = e.getX() - startX;
    double draggedDistanceY = e.getY() - startY;

    // Reset the starting point for the next drag
    // if the user keeps the mouse pressed and drags
again
    startX = e.getX();
    startY = e.getY();

    // Get the minX and minY of the current viewport
    double curMinX = imageView.getViewport().getMinX();
    double curMinY = imageView.getViewport().getMinY();

    // Move the new viewport by the dragged distance
    double newMinX = curMinX + draggedDistanceX;
    double newMinY = curMinY + draggedDistanceY;

    // Make sure the viewport does not fall outside the
image area
}

```

```

        newMinX = clamp(newMinX, 0,
imageView.getImage().getWidth() - VIEWPORT_WIDTH);
        newMinY = clamp(newMinY, 0,
imageView.getImage().getHeight() - VIEWPORT_HEIGHT);

        // Set a new viewport
        imageView.setViewport(
            new Rectangle2D(newMinX, newMinY,
VIEWPORT_WIDTH, VIEWPORT_HEIGHT));
    }

    private double clamp(double value, double min, double max)
{
    if (value < min) {
        return min;
    } else if (value > max) {
        return max;
    }

    return value;
}
}

```

The program declares a few class and instance variables.

The `VIEWPORT_WIDTH` and `VIEWPORT_HEIGHT` are constants holding the width and height of the viewport. The `startX` and `startY` instance variables will hold the x and y coordinates of the mouse when the mouse is pressed or dragged. The `ImageView` instance variable holds the reference of the `ImageView`. We need this reference in the mouse dragged event handler.

The starting part of the `start()` method is simple. It creates an `Image`, an `ImageView`, and sets a viewport for the `ImageView`. Then, it sets the mouse pressed and dragged event handlers to the `ImageView`.

```
// Set the mouse pressed and mouse dradded event hanlders
imageView.setOnMousePressed(this::handleMousePressed);
imageView.setOnMouseDragged(this::handleMouseDragged);
```

In the `handleMousePressed()` method, we store the coordinates of the mouse in the `startX` and `startY` instance variables. The coordinates are relative to the `ImageView`.

```
startX = e.getX();
startY = e.getY();
```

The `handleMousePressed()` method computes the new location of the viewport inside the image because of the mouse drag and sets a new viewport at the new location. First, it computes the dragged distance for the mouse along the x-axis and y-axis.

```
// How far the mouse was dragged
double draggedDistanceX = e.getX() - startX;
double draggedDistanceY = e.getY() - startY;
```

You reset the `startX` and `startY` values to mouse location that triggered the current mouse dragged event. This is important to get the correct dragged distance when the user keeps the mouse pressed, drags it, stops without releasing the mouse, and drags it again.

```
// Reset the starting point for the next drag
// if the user keeps the mouse pressed and drags again
startX = e.getX();
startY = e.getY();
```

You compute the new location of the upper-left corner of the viewport. You always have a viewport in the `ImageView`. The new viewport will be located at the dragged distance from the old location.

```
// Get the minX and minY of the current viewport
double curMinX = imageView.getViewport().getMinX();
double curMinY = imageView.getViewport().getMinY();

// Move the new viewport by the dragged distance
double newMinX = curMinX + draggedDistanceX;
double newMinY = curMinY + draggedDistanceY;
```

It is fine to place the viewport outside the region of the image. The viewport simply displays an empty area when it falls outside the image area. To restrict the viewport inside the image area, we clamp the location of the viewport.

```
// Make sure the viewport does not fall outside the image area
newMinX = clamp(newMinX, 0, imageView.getImage().getWidth() -
    VIEWPORT_WIDTH);
newMinY = clamp(newMinY, 0, imageView.getImage().getHeight() -
    VIEWPORT_HEIGHT);
```

Finally, we set a new viewport using the new location.

```
// Set a new viewport
imageView.setViewport(new Rectangle2D(newMinX, newMinY,
    VIEWPORT_WIDTH, VIEWPORT_HEIGHT));
```

**Tip** It is possible to scale or rotate the `ImageView` and set a viewport to view the region of the image defined by the viewport.

## Understanding Image Operations

JavaFX supports reading pixels from an image, writing pixels to an image, and creating a snapshot of the scene. It supports creating an image from scratch. If an image is writable, you can also modify the image in memory and save it to the file system. The image API provides access to each pixel in the image. It supports reading and writing one pixel or a chunk of pixel at a time. This section will discuss operations supported by the image API with simple examples.

### Pixel Formats

The image API in JavaFX gives you access to each pixel in an image. A pixel stores information about its color (red, green, blue) and opacity (alpha). The pixel information can be stored in several formats.

An instance of the `PixelFormat<T extends Buffer>` represents the layout of data for a pixel. You need to know the pixel format when you read the pixels from an image. You need to specify the pixel format when you write pixels to an image. The `WritablePixelFormat` class inherits from the `PixelFormat` class and its instance represents a pixel format that can store full color information. An instance of the `WritablePixelFormat` class is used when writing pixels to an image.

Both class `PixelFormat` and its subclass `WritablePixelFormat` are abstract. The `PixelFormat` class provides several static methods to obtain instances to `PixelFormat` and `WritablePixelFormat` abstract classes. Before we discuss how to get an instance of the `PixelFormat`, let us discuss types of storage formats available for storing the pixel data.

A `PixelFormat` has a type that specifies the storage format for a single pixel. The constants of the `PixelFormat.Type` enum represent different type of storage formats:

- `BYTE_RGB`
- `BYTE_BGRA`
- `BYTE_BGRA_PRE`
- `BYTE_INDEXED`
- `INT_ARGB`
- `INT_ARGB_PRE`

In the `BYTE_RGB` format, the pixels are assumed opaque. The pixels are stored in adjacent bytes as red, green, and blue, in order.

In the `BYTE_BGRA` format, pixels are stored in adjacent bytes as blue, green, red, and alpha in order. The color values (red, green, and blue) are not pre-multiplied with the alpha value.

The `BYTE_BGRA_PRE` type format is similar to `BYTE_BGRA`, except that in `BYTE_BGRA_PRE` the stored color component values are pre-multiplied by the alpha value.

In the `BYTE_INDEXED` format, a pixel is as a single byte. A separate lookup list of colors is provided. The single byte value for the pixel is used as an index in the lookup list to get the color value for the pixel.

In the `INT_ARGB` format, each pixel is stored in a 32-bit integer. Bytes from the most significant byte (MSB) to the least significant byte (LSB) store alpha, red, green, and blue values. The color values (red, green, and

blue) are not pre-multiplied with the alpha value. The following snippet of code shows how to extract components from a pixel value in this format.

```
int pixelValue = get the value for a pixel...
int alpha = (pixelValue >> 24) & 0xff;
int red   = (pixelValue >> 16) & 0xff;
int green = (pixelValue >> 8) & 0xff;
int blue  = pixelValue & 0xff;
```

The `INT_ARGB_PRE` format is similar to the `INT_ARGB` format, except that `INT_ARGB_PRE` stores the color values (red, green, and blue) pre-multiplied with the alpha value.

Typically, you need to create a `WritablePixelFormat` when you write pixels to create a new image. When you read pixels from an image, the pixel reader will provide you a `PixelFormat` instance that will tell you how the color information in the pixels are stored. The following snippet of code creates some instances of `WritablePixelFormat` class:

```
import javafx.scene.image.PixelFormat;
import javafx.scene.image.WritablePixelFormat;
import java.nio.ByteBuffer;
import java.nio.IntBuffer;

...
// BYTE_BGRA Format type
WritablePixelFormat<ByteBuffer> format1
= PixelFormat.getByteBgraInstance();

// BYTE_BGRA_PRE Format type
WritablePixelFormat<ByteBuffer> format2
= PixelFormat.getByteBgraPreInstance();

// INT_ARGB Format type
WritablePixelFormat<IntBuffer> format3
= PixelFormat.getIntArgbInstance();

// INT_ARGB_PRE Format type
WritablePixelFormat<IntBuffer> format4
= PixelFormat.getIntArgbPreInstance();
```

**Pixel format classes** are not useful without pixel information. After all, they describe layout of information in a pixel! We will use these classes when we read and write image pixels in the sections to follow. Their use will be obvious in the examples.

## Reading Pixels from an Image

An instance of the `PixelReader` interface is used to read pixels from an image. Use the `getPixelReader()` method of the `Image` class to obtain a `PixelReader`. The `PixelReader` interface contains the following methods:

- int getArgb(int x, int y)
- Color getColor(int x, int y)
- Void getPixels(int x, int y, int w, int h, WritablePixelFormat<ByteBuffer> pixelformat, byte[] buffer, int offset, int scanlineStride)
- void getPixels(int x, int y, int w, int h, WritablePixelFormat<IntBuffer> pixelformat, int[] buffer, int offset, int scanlineStride)
- <T extends Buffer> void getPixels(int x, int y, int w, int h, WritablePixelFormat<T> pixelformat, T buffer, int scanlineStride)
- PixelFormat getPixelFormat()

The `PixelReader` interface contains methods to read one pixel or multiple pixels at a time. Use the `getArgb()` and `getColor()` methods to read the pixel at the specified (x, y) coordinate. Use the `getPixels()` method to read pixels in bulk. Use the `getPixelFormat()` method to get the `PixelFormat` that best describes the storage format for the pixels in the source.

The `getPixelReader()` method of the `Image` class returns a `PixelReader` only if the image is readable. Otherwise, it returns null. An image may not be readable if it is not fully loaded yet, it had an error during loading, or its format does not support reading pixels.

```
Image image = new Image("resources/picture/ksharan.jpg");

// Get the pixel reader
PixelReader pixelReader = image.getPixelReader();
if (pixelReader == null) {
    System.out.println("Connot read pixels from the image");
} else {
    // Read image pixels
}
```

Once you have a `PixelReader`, you can read pixels invoking one of its methods. The program in Listing 24-4 shows how to read pixels from an image. The code is self-explanatory.

- The `start()` method creates an `Image`. The `Image` is loaded synchronously.
- The logic to read the pixels is in the `readPixelsInfo()` method. The method receives a fully loaded `Image`. It uses the `getColor()` method of

the `PixelReader` to get the pixel at a specified location. It prints the colors for all pixels. At the end, it prints the pixel format, which is `BYTE_RGB`.

### ***Listing 24-4.*** Reading Pixels from an Image

```
// ReadPixelInfo.java
package com.jdojo.image;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.image.PixelFormat;
import javafx.scene.image.PixelReader;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class ReadPixelInfo extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        String imagePath = "resources/picture/ksharan.jpg";
        Image image = new Image(imagePath);
        ImageView imageView = new ImageView(image);
        HBox root = new HBox(imageView);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Reading Pixels from an Image");
        stage.show();

        // Read pixels from the image
        this.readPixelsInfo(image);
    }

    private void readPixelsInfo(Image image) {
        // Obtain the pixel reader from the image
        PixelReader pixelReader = image.getPixelReader();
        if (pixelReader == null) {
            System.out.println("Connot read pixels from the
image");
            return;
        }

        // Get image width and height
        int width = (int)image.getWidth();
        int height = (int)image.getHeight();

        // Read all pixels
    }
}
```

```

        for(int y = 0; y < height; y++) {
            for(int x = 0; x < width; x++) {
                Color color = pixelReader.getColor(x, y);
                System.out.println("Color at (" + x + ", "
" + y + ") = " + color);
            }
        }

        PixelFormat format = pixelReader.getPixelFormat();
        PixelFormat.Type formatType = format.getType();
        System.out.println("Pixel format type: "
+ formatType);
    }
}
Color at (0, 0) = 0xb5bb41ff
Color at (1, 0) = 0xb0b53dff
...
Color at (233, 287) = 0x718806ff
Color at (234, 287) = 0x798e0bff
Pixel format type: BYTE_RGB

```

Reading pixels in bulk is little more difficult than reading one pixel at a time. The difficulty arises from the setup information that you have to provide to the `getPixels()` method. We will repeat the above example by reading all pixels in bulk using the following method of the `PixelReader`.

```

void getPixels(int x,
               int y,
               int width, int height,
               WritablePixelFormat<ByteBuffer> pixelformat,
               byte[] buffer,
               int offset,
               int scanlineStride)

```

The method reads the pixels from rows in order. The pixels in the first row are read, then the pixels from the second row, and so on. It is important that you understand the meaning of all parameters to the method:

The method reads the pixels of a rectangular region in the source.

The `x` and `y` coordinates of the upper-left corner of the rectangular region are specified in the `x` and `y` arguments.

The `width` and `height` arguments specify the width and height of the rectangular region.

The `pixelformat` specifies the format of the pixel that should be used to store the read pixels in the specified `buffer`.

The `buffer` is a `byte array` in which the `PixelReader` will store the read pixels. The length of the array must be big enough to store all read pixels.

The `offset` specifies the starting index in the `buffer` array to store the first pixel data. Its value of zero indicates that the data for the first pixel will start at index 0 in the buffer.

The `scanlineStride` specify the distance between the start of one row of data In the buffer to the start of the next row of data. Suppose you have two pixels in a row and you want to read in the `BYTE_BGRA` format taking four bytes for a pixel. One row of data can be stored in eight bytes. If you specify 8 as the argument value, the data for the next row will start in the buffer just after the data for the previous row data ends. If you specify the argument value 10, last two bytes will be empty for each row of data. The first row pixels will be stored from index 0 to 7. The indexes 8 and 9 will be empty (or not written). Indexes 10 to 17 will store pixel data for the second row leaving indexes 18 and 19 empty. You may want to specify a bigger value for the agrument than needed to store one row of pixel data if you want to fill the empty slots with your own values later. Specifying avlaue less than needed will overwrite part of the data in the previous row.

The following snippet of code shows hwo to read all pixels from an image in a byte array.in `BYTE_BGRA` format.

```
Image image = ...
PixelReader pixelReader = image.getPixelReader();

int x = 0;
int y = 0;
int width = (int)image.getWidth();
int height = (int)image.getHeight();
int offset = 0;
int scanlineStride = width * 4;
byte[] buffer = new byte[width * height * 4];

// Get a WritablePixelFormat for the BYTE_BGRA format type
WritablePixelFormat<ByteBuffer> pixelFormat
= PixelFormat.getByteBgraInstance();

// Read all pixels at once
pixelReader.getPixels(x, y,
                      width, height,
                      pixelFormat,
                      buffer,
                      offset,
                      scanlineStride);
```

The x and y coordinates of the upper-left corner of the rectangular region to be read are set to zero. The width and height of the region are set to the width and height of the image. This sets up the arguments to read the entire image.

You want to read the pixel data into the buffer starting at index 0, so you set the `offset` argument to 0.

You want to read the pixel data in `BYTE_BGRA` format type, which takes 4 bytes to store data for one pixel. We have set the `scanlineStride` argument value, which is the length of a row data, to `width * 4`, so a row data starts at the next index from where the previous row data ended.

You get an instance of the `WritablePixelFormat` to read the data in the `BYTE_BGRA` format type. Finally, we call the `getPixels()` method of the `PixelReader` to read the pixel data. The buffer will be filled with the pixel data when the `getPixels()` method returns.

**Tip** Setting the value for the `scanlineStride` argument and the length of the buffer array depends on the `pixelFormat` argument. Other versions of the `getPixels()` method allows reading pixel data in different formats.

The program in Listing 24-5 has the complete source code to read pixels in bulk. After reading all pixels, it decodes the color components in the byte array for the pixel at  $(0, 0)$ . It reads the pixel at  $(0, 0)$  using the `getColor()` method. The pixel data at  $(0, 0)$  obtained through both methods are printed on the standard output.

### ***Listing 24-5.*** Reading Pixels from an Image in Bulk

```
// BulkPixelReading.java
package com.jdojo.image;

import java.nio.ByteBuffer;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.image.PixelFormat;
import javafx.scene.image.PixelReader;
import javafx.scene.image.WritablePixelFormat;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class BulkPixelReading extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        String imagePath = "resources/picture/ksharan.jpg";
        Image image = new Image(imagePath);
        ImageView imageView = new ImageView(image);
```

```

HBox root = new HBox(imageView);
Scene scene = new Scene(root);
stage.setScene(scene);
stage.setTitle("Reading Pixels in Bulk");
stage.show();

// Read pixels in bulk from the image
this.readPixelsInfo(image);
}

private void readPixelsInfo(Image image) {
    // Obtain the pixel reader from the image
    PixelReader pixelReader = image.getPixelReader();
    if (pixelReader == null) {
        System.out.println("Connot read pixels from the
image");
        return;
    }

    // Read all pixels in a byte array in one go
    int x = 0;
    int y = 0;
    int width = (int)image.getWidth();
    int height = (int)image.getHeight();
    int offset = 0;
    int scanlineStride = width * 4;
    byte[] buffer = new byte[width * height * 4];

    // Get a WritablePixelFormat
    WritablePixelFormat<ByteBuffer> pixelFormat
= PixelFormat.getByteBgraInstance();

    // Read all pixels at once
    pixelReader.getPixels(x, y,
                          width, height,
                          pixelFormat,
                          buffer,
                          offset,
                          scanlineStride);

    // Read the color of the pixel at (0, 0)
    int blue = (buffer[0] & 0xff);
    int green = (buffer[1] & 0xff);
    int red = (buffer[2] & 0xff);
    int alpha = (buffer[3] & 0xff);
    System.out.println("red=" + red + ", green=" + green
+
", blue=" + blue + ", alpha="
+ alpha);

    // Get the color of the pixel at (0, 0)
    Color c = pixelReader.getColor(0, 0);
    System.out.println("red=" + (int)(c.getRed() * 255) +
",
green=" + (int)(c.getGreen() * 255)
+

```

```

        ", blue=" + (int)(c.getBlue() * 255) +
        ", alpha=" + (int)(c.getOpacity()
* 255));
    }
}
red=181, green=187, blue=65, alpha=255
red=181, green=187, blue=65, alpha=255

```

## Writing Pixels to an Image

You can write pixels to an image or any surface that supports writing pixels. For example, you can write pixels to a `WritableImage` and a `Canvas`.

**Tip** An `Image` is a read-only pixel surface. You can read pixels from an `Image`. However, you cannot write pixels to an `Image`. If you want to write to an image or create an image from scratch, use a `WritableImage`.

An instance of the `PixelWriter` interface is used to write pixels to a surface. A `PixelWriter` is provided by the writable surface. For example, you can use the `getPixelWriter()` method of the `Canvas` and `WritableImage` to obtain a `PixelWriter` for them.

The `PixelWriter` interface contains methods to write pixels to a surface and obtain the pixel format supported by the surface:

- `PixelFormat getPixelFormat()`
- `void setArgb(int x, int y, int argb)`
- `void setColor(int x, int y, Color c)`
- `void setPixels(int x, int y, int w, int h, PixelFormat<ByteBuffer> pixelformat, byte[] buffer, int offset, int scanlineStride)`
- `void setPixels(int x, int y, int w, int h, PixelFormat<IntBuffer> pixelformat, int[] buffer, int offset, int scanlineStride)`
- `<T extends Buffer> void setPixels(int x, int y, int w, int h, PixelFormat<T> pixelformat, T buffer, int scanlineStride)`
- `void setPixels(int dstx, int dsty, int w, int h, PixelReader reader, int srcx, int srcy)`

The `getPixelFormat()` method returns the pixel format in which the pixels can be written to the surface.

The `setArgb()` and `setColor()` methods allow for writing one pixel at the specified (x, y) location in the destination surface.

The `setArgb()` method accepts the pixel data in an integer in the

INT\_ARGB format whereas the `setColor()` method accepts a `Color` object. The `setPixels()` methods allow for bulk pixel writing.

You can use an instance of the `WritableImage` to create an image from scratch. The class contains three constructors:

- `WritableImage(int width, int height)`
- `WritableImage(PixelReader reader, int width, int height)`
- `WritableImage(PixelReader reader, int x, int y, int width, int height)`

The first constructor creates an empty image of the specified width and height.

```
// Create a new empty image of 200 x 100
WritableImage newImage = new WritableImage(200, 100);
```

The second constructor creates an image of the specified width and height. The specified reader is used to fill the image with pixels. An `ArrayIndexOutOfBoundsException` is thrown if the reader reads from a surface that does not have the necessary number of rows and columns to fill the new image. Use this constructor to copy the whole or part of an image. The following snippet of code creates a copy of an image.

```
String imagePath = "resources/picture/ksharan.jpg";
Image image = new Image(imagePath, 200, 100, true, true);

int width = (int)image.getWidth();
int height = (int)image.getHeight();

// Create a copy of the image
WritableImage newImage = new
WritableImage(image.getPixelReader(), width, height);
```

The third constructor lets you copy a rectangular region from a surface. The `(x, y)` value is coordinates of the upper-left corner of the rectangular region. The `(width, height)` value is the dimension of the rectangular region to be read using the reader and the desired dimension of the new image.

An `ArrayIndexOutOfBoundsException` is thrown if the reader reads from a surface that does not have the necessary number of rows and columns to fill the new image.

The `WritableImage` is a read-write image.

Its `getPixelWriter()` method returns a `PixelWriterto` write pixels to the image. It inherits the `getPixelReader()` method that returns a `PixelReaderto` read data from the image.

The following snippet of code creates an `Image` and an empty `WritableImage`. It reads one pixel at a time from the `Image`, makes the pixel darker, and writes the same pixel to the new `WritableImage`. At the end, we have created a darker copy of the original image.

```
Image image = new Image("resources/picture/ksharan.jpg");
PixelReader pixelReader = image.getPixelReader();
int width = (int)image.getWidth();
int height = (int)image.getHeight();

// Create a new, empty WritableImage
WritableImage darkerImage = new WritableImage(width, height);
PixelWriter darkerWriter = darkerImage.getPixelWriter();

// Read one pixel at a time from the source and
// write it to the destinations - one darker and one brighter
for(int y = 0; y < height; y++) {
    for(int x = 0; x < width; x++) {
        // Read the pixel from the source image
        Color color = pixelReader.getColor(x, y);

        // Write a darker pixel to the new image at the same
        location
        darkerWriter.setColor(x, y, color.darker());
    }
}
```

The program in Listing 24-6 creates an `Image`. It creates three instances of the `WritableImage` and copies the pixels from the original image to them. The copied pixels are modified before they written to the destination. For one destination, pixels are darkened: for one brightened, and for one, made semi-transparent. All four images are displayed in `ImageViews` as shown in Figure 24-4.

### ***Listing 24-6.*** Writing Pixels to an Image

```
// CopyingImage.java
package com.jdojo.image;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.image.PixelReader;
import javafx.scene.image.PixelWriter;
import javafx.scene.image.WritableImage;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.text.Text;
import javafx.stage.Stage;
```

```

public class CopyingImage extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        String imagePath = "resources/picture/ksharan.jpg";
        Image image = new Image(imagePath, 200, 100, true,
true);

        int width = (int)image.getWidth();
        int height = (int)image.getHeight();

        // Create three WritableImage instances
        // one will be a darker, one brighter, and one semi-
transparent
        WritableImage darkerImage = new WritableImage(width,
height);
        WritableImage brighterImage = new
WritableImage(width, height);
        WritableImage semiTransparentImage = new
WritableImage(width, height);

        // Copy source pixels to the destinations
        this.createImages(image,
            darkerImage,
            brighterImage,
            semiTransparentImage,
            width,
            height);

        ImageView imageView = new ImageView(image);
        ImageView darkerView = new ImageView(darkerImage);
        ImageView brighterView = new
ImageView(brighterImage);
        ImageView semiTransparentView = new
ImageView(semiTransparentImage);

        HBox root = new HBox(10,
            new VBox(imageView, new Text("Original")),
            new VBox(darkerView, new Text("Darker")),
            new VBox(brighterView, new Text("Brighter")),
            new VBox(semiTransparentView, new Text("Semi-
Transparent")));
    }

    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("Writing Pixels to an Image");
    stage.show();
}

private void createImages(Image image,
    WritableImage darkerImage,

```

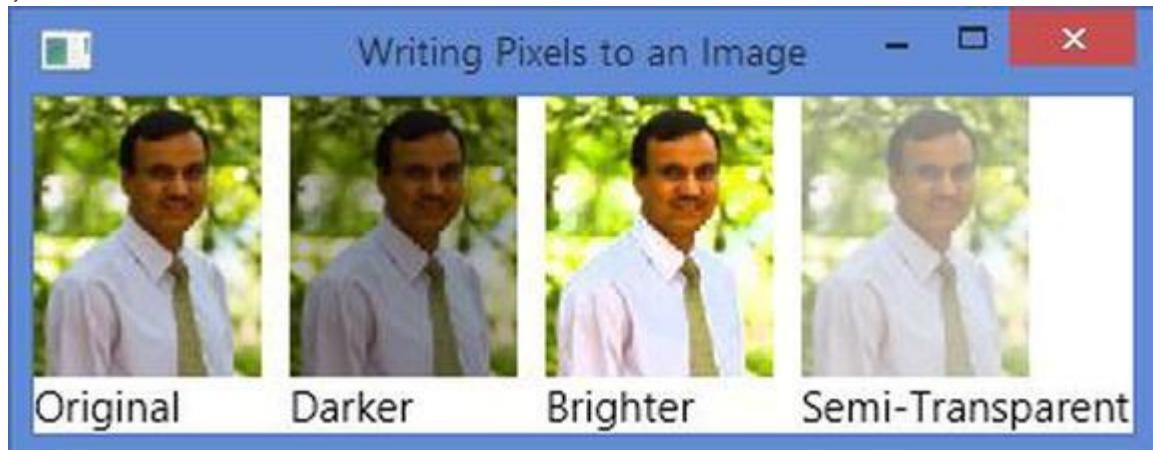
```
WritableImage brighterImage,
WritableImage semiTransparentImage,
int width, int height) {
    // Obtain the pixel reader from the image
    PixelReader pixelReader = image.getPixelReader();
    PixelWriter darkerWriter = darkerImage.getPixelWriter();
    PixelWriter brighterWriter
= brighterImage.getPixelWriter();
    PixelWriter semiTransparentWriter
= semiTransparentImage.getPixelWriter();

    // Read one pixel at a time from the source and
    // write it to the destinations
    for(int y = 0; y < height; y++) {
        for(int x = 0; x < width; x++) {
            Color color = pixelReader.getColor(x, y);

            // Write a darker pixel to the new image
            darkerWriter.setColor(x, y,
color.darker());

            // Write a brighter pixel to the new
image
            brighterWriter.setColor(x, y,
color.brighter());

            // Write a semi-transparent pixel to the
new image
            semiTransparentWriter.setColor(x, y,
                Color.color(color.getRed(),
color.getGreen(),
color.getBlue(),
0.50));
        }
    }
}
```



**Figure 24-4.** Original image and modified images

**Tip** It is easy to crop an image in JavaFX. Use one of the `getPixels()` methods of the `PixelReader` to read the needed area of the image in a buffer and write the buffer to a new image. This gives you a new image that is the cropped version of the original image.

## Creating an Image from Scratch

In the previous section, we created new images by copying pixels from another image. We had altered the color and opacity of the original pixels before writing them to the new image. That was easy because we were working on one pixel at a time and we received a pixel as a `Color` object. It is also possible to create pixels from scratch then use them to create a new image. Anyone would admit that creating a new, meaningful image by defining its each pixel in code is not an easy task. However, JavaFX has made the process of doing so easy.

In this section, we will create a new image with a pattern of rectangles placed in a grid-like fashion. Each rectangle will be divided into two parts using the diagonal connecting the upper-left and lower-right corners. The upper triangle is painted in green and the lower in red. A new image will be created and filled with the rectangles.

Creating an image from scratch involves three steps:

- Create an instance of the `WritableImage`.
- Create buffer (a `byte` array, an `int` array, etc.) and populate it with pixel data depending on the pixel format you want to use for the pixels data.
- Write the pixels in the buffer to the image.

Let us write the code that creates the pixels for our rectangular region. Let us declare constants for the width and height of the rectangle.

```
static final int RECT_WIDTH = 20;
static final int RECT_HEIGHT = 20;
```

We need to define a buffer (a `byte` array) big enough to hold data for all pixels. Each pixel in `BYTE_RGB` format takes 2 bytes.

```
byte[] pixels = new byte[RECT_WIDTH * RECT_HEIGHT * 3];
```

If the region is rectangular, we need to know the height to width ratio to divide the region into upper and lower rectangles.

```
double ratio = 1.0 * RECT_HEIGHT/RECT_WIDTH;
```

The following snippet of code populates the buffer.

```
// Generate pixel data
for (int y = 0; y < RECT_HEIGHT; y++) {
    for (int x = 0; x < RECT_WIDTH; x++) {
        int i = y * RECT_WIDTH * 3 + x * 3;
        if (x <= y/ratio) {
            // Lower-half
```

```

    pixels[i] = -1; // red -1 means 255 (-1 & 0xff
= 255)
        pixels[i+1] = 0; // green = 0
        pixels[i+2] = 0; // blue = 0
    } else {
        // Upper-half
        pixels[i] = 0; // red = 0
        pixels[i+1] = -1; // Green 255
        pixels[i+2] = 0; // blue = 0
    }
}
}

```

Pixels are stored in the buffer in the row first order. The variable `i` inside the loop computes the position in the buffer where the 3-byte data starts for a pixel. For example, the data for the pixel at  $(0, 0)$  starts at the index 0; the data for the pixel at  $(0, 1)$  starts at index 3, etc. The 3 bytes for a pixel stores red, green, and blue values in order of increasing index. Encoded values for the color components are stored in the buffer, so that the expression “`byteValue & 0xff`” will produce the actual color component value between 0 and 255. If you want a red pixel, you need to set -1 for the red component as “`-1 & 0xff`” produces 255. For a red color, the green and blue components will be set to zero. A byte array initializes all elements to zero. However, we have explicitly set them to zero in our code. For the lower-half triangle, we set the color to green. The condition “`x = <= y/ratio`” is used to determine the position of a pixel whether it falls in the upper-half triangle or the lower-half triangle. If the `y/ratio` is not an integer, the division of the rectangle into two triangles may be a little off at the lower-right corner.

Once we get the pixel data, we need to write them to a `WritableImage`. The following snippet of code writes the pixels for the rectangle, once at the upper-left corner of the image.

```
WritableImage newImage = new WritableImage(350, 100);
PixelWriter pixelWriter = newImage.getPixelWriter();
byte[] pixels = generate pixel data...
```

```
// Our data is in BYTE_RGB format
PixelFormat<ByteBuffer> pixelFormat
= PixelFormat.getByteRgbInstance();
Int xPos = 0;
int yPos = 0;
int offset = 0;
int scanlineStride = RECT_WIDTH * 3;
pixelWriter.setPixels(xPos, yPos,
                      RECT_WIDTH, RECT_HEIGHT,
                      pixelFormat,
                      pixels, offset,
                      scanlineStride);
```

The program in Listing 24-7 creates an image from scratch. It creates a pattern by writing row pixels for the rectangular region to fill the image. Figure 24-5 shows the image.

### ***Listing 24-7.*** Creating an Image from Scratch

```
// CreatingImage.java
package com.jdojo.image;

import java.nio.ByteBuffer;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.image.ImageView;
import javafx.scene.image.PixelFormat;
import javafx.scene.image.PixelWriter;
import javafx.scene.image.WritableImage;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class CreatingImage extends Application {
    private static final int RECT_WIDTH = 20;
    private static final int RECT_HEIGHT = 20;

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

    @Override
    public void start(Stage stage) {
        WritableImage newImage = new WritableImage(350, 100);

        // Get the pixels data
        byte[] pixels = getPixelsData();

        // Write pixels data to the image
        this.writePattern(newImage, pixels);

        // Display the new image in an ImageView
        ImageView newImageView = new ImageView(newImage);

        HBox root = new HBox(newImageView);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Creating an Image from Scratch");
        stage.show();
    }

    private byte[] getPixelsData() {
        // Each pixel takes 3 bytes
        byte[] pixels = new byte[RECT_WIDTH * RECT_HEIGHT
        * 3];

        // Height to width ratio
        double ratio = 1.0 * RECT_HEIGHT/RECT_WIDTH;
```

```

        // Generate pixels data
        for (int y = 0; y < RECT_HEIGHT; y++) {
            for (int x = 0; x < RECT_WIDTH; x++) {
                int i = y * RECT_WIDTH * 3 + x * 3;
                if (x <= y/ratio) {
                    pixels[i] = -1; // red -1 means
255 (-1 & 0xff = 255)
                    pixels[i+1] = 0; // green = 0
                    pixels[i+2] = 0; // blue = 0
                } else {
                    pixels[i] = 0; // red = 0
                    pixels[i+1] = -1; // Green 255
                    pixels[i+2] = 0; // blue = 0
                }
            }
        }
        return pixels;
    }

    private void writePattern(ReadableImage newImage, byte[]
pixels) {
    PixelWriter pixelWriter = newImage.getPixelWriter();

    // Our data is in BYTE_RGB format
    PixelFormat<ByteBuffer> pixelFormat
= PixelFormat.getByteRgbInstance();

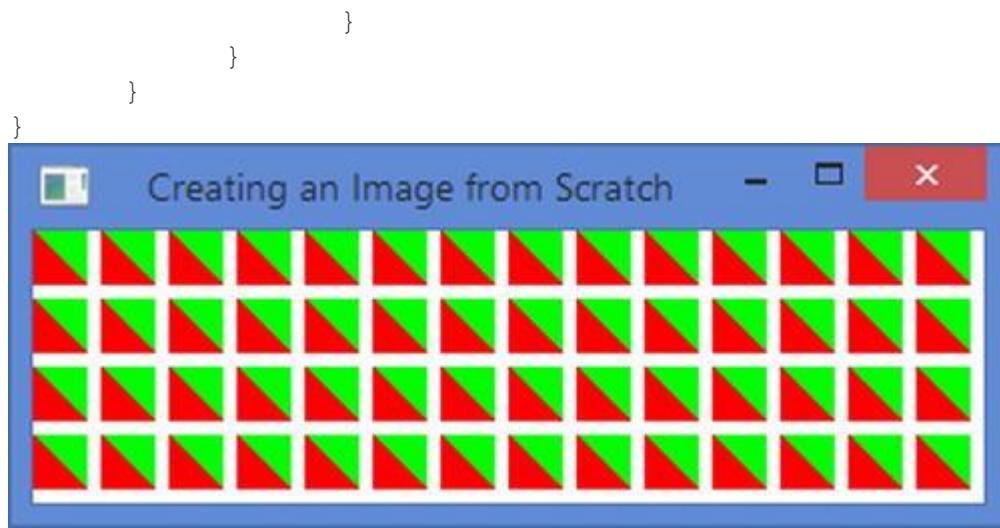
    int spacing = 5;
    int imageWidth = (int)newImage.getWidth();
    int imageHeight = (int)newImage.getHeight();

    // Roughly compute the number of rows and columns
    int rows = imageHeight/(RECT_HEIGHT + spacing);
    int columns = imageWidth/(RECT_WIDTH + spacing);

    // Write the pixels to the image
    for (int y = 0; y < rows; y++) {
        for (int x = 0; x < columns; x++) {
            // Compute the current location inside
the image where
            // the rectangular region to be written
            int xPos = x * (RECT_WIDTH + spacing);
            int yPos = y * (RECT_HEIGHT + spacing);

            // Write the pixels data at the current
location
            // defined by xPos and yPos
            pixelWriter.setPixels(xPos, yPos,
                RECT_WIDTH,
RECT_HEIGHT,
                pixelFormat,
                pixels, 0,
                RECT_WIDTH * 3);
        }
    }
}

```



**Figure 24-5.** An image created from scratch

### Saving a New Image to a FileSystem

Saving an `Image` to the file system is easy:

- Convert the `Image` to a `BufferedImage` using the `fromFXImage()` method of the `SwingFXUtils` class.
- Pass the `BufferedImage` to the `write()` method of the `ImageIO` class.

Notice that we have to use two classes – `BufferedImage` and `ImageIO` – that are part of the standard Java library, not the JavaFX library. The following snippet of code shows the outline of the steps involved in saving an image to a file in the PNG format.

```
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javafx.embed.swing.SwingFXUtils;
import javafx.scene.image.Image;
import javax.imageio.ImageIO;
...

Image image = create an image...
BufferedImage bImage = SwingFXUtils.fromFXImage(image, null);

// Save the image to the file
File fileToSave = ...
String imageFormat = "png";
try {
    ImageIO.write(bImage, imageFormat, fileToSave);
}
catch (IOException e) {
```

```

        throw new RuntimeException(e);
    }
}

```

The program in Listing 24-8 has code for a utility class `ImageUtil`. Its static `saveToFile(Image image)` method can be used to save an `Image` to a local file system. The method asks for a file name. The user can select a PNG or a JPEG format for the image.

### ***Listing 24-8.*** A Utility Class to Save an Image to a File

```

// ImageUtil.java
package com.jdojo.image;

import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javafx.embed.swing.SwingFXUtils;
import javafx.scene.image.Image;
import javafx.stage.FileChooser;
import static javafx.stage.FileChooser.ExtensionFilter;
import javax.imageio.ImageIO;

public class ImageUtil {
    public static void saveToFile(Image image) {
        // Ask the user for the file name
        FileChooser fileChooser = new FileChooser();
        fileChooser.setTitle("Select an image file name");
        fileChooser.setInitialFileName("untitled");
        ExtensionFilter pngExt = new ExtensionFilter("PNG
Files", "*.png");
        ExtensionFilter jpgExt =
            new ExtensionFilter("JPEG Files",
"*.jpg", "*.jpeg");
        fileChooser.getExtensionFilters().addAll(pngExt,
jpgExt);

        File outputFile = fileChooser.showSaveDialog(null);
        if (outputFile == null) {
            return;
        }

        ExtensionFilter selectedExt
= fileChooser.getSelectedExtensionFilter();
        String imageFormat = "png";
        if (selectedExt == jpgExt) {
            imageFormat = "jpg";
        }

        // Check for the file extension. Add oen, iff not
specified
        String fileName = outputFile.getName().toLowerCase();
        switch (imageFormat) {
            case "jpg":

```

```

                if (!fileName.endsWith(".jpeg") &&
!fileName.endsWith(".jpg")) {
                    outputFile = new
File(outputFile.getParentFile(),
                                outputFile.getName
() + ".jpg");
                }
            break;
        case "png":
            if (!fileName.endsWith(".png")) {
                outputFile = new
File(outputFile.getParentFile(),
                                outputFile.getName
() + ".png");
            }
        }

        // Convert the image to a buffered image
        BufferedImage bImage
= SwingFXUtils.fromFXImage(image, null);

        // Save the image to the file
        try {
            ImageIO.write(bImage, imageFormat, outputFile);
        }
        catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
}

```

The program in Listing 24-9 shows how to save an image to a file. Click the Save Image button to save the picture to a file. It opens a file chooser dialog to let you select a file name. If you cancel the file chooser dialog, the saving process is aborted.

### ***Listing 24-9.*** Saving an Image to a File

```

// SaveImage.java
package com.jdojo.image;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class SaveImage extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }
}

```

```
    @Override
    public void start(Stage stage) {
        String imagePath = "resources/picture/ksharan.jpg";
        Image image = new Image(imagePath);
        ImageView imageView = new ImageView(image);

        Button saveBtn = new Button("Save Image");
        saveBtn.setOnAction(e ->
ImageUtil.saveToFile(image));

        VBox root = new VBox(10, imageView, saveBtn);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Saving an Image to a File");
        stage.show();
    }
}
```

## Taking the Snapshot of a Node and a Scene

JavaFX allows you to take a snapshot of a `Node` and a `Scene` as they will appear in the next frame. You get the snapshot in a `WritableImage`, which means you can perform all pixel-level operations after you take the snapshot. The `Node` and `Scene` classes contain a `snapshot()` method to accomplish this.

### Taking the Snapshot of a Node

The `Node` class contains an overloaded `snapshot()` method:

- `WritableImage snapshot(SnapshotParameters params, WritableImage image)`
- `void snapshot(Callback<SnapshotResult,Void> callback, SnapshotParameters params, WritableImage image)`

The first version of the `snapshot()` method is synchronous whereas the second one is asynchronous. The method lets you specify an instance of the `SnapshotParameters` class that contains the rendering attributes for the snapshot. If this is null, default values will be used. You can set the following attributes for the snapshot:

- A fill color
- A transform
- A viewport
- A camera
- A depth buffer

By default, the fill color is white; no transform and viewport are used; a ParallelCamera is used; and, the depth buffer is set to false. Note that these attributes are used on the node only while taking its snapshot.

You can specify a WritableImage in the `snapshot()` method that will hold the snapshot of the node. If this is null, a new WritableImage is created. If the specified WritableImage is smaller than the node, the node will be clipped to fit the image size.

The first version of the `snapshot()` method returns the snapshot in a WritableImage. The image is either the one that is passed as the parameter or a new one created by the method.

The second, asynchronous version of the `snapshot()` method accepts a `Callback` object whose `call()` method is called. A `SnapshotResult` object is passed to the `call()` method, which can be used to obtain the snapshot image, the source node, and the snapshot parameters using the following methods:

- `WritableImage getImage()`
- `SnapshotParameters getSnapshotParameters()`
- `Object getSource()`

**Tip** The `snapshot()` method takes the snapshot of the node using the `boundsInParent` property of the node. That is, the snapshot contains all effects and transformations applied to the node. If the node is being animated, the snapshot will include the animated state of the node at the time it is taken.

The program in Listing 24-10 shows how to take a snapshot of a `TextField` node. It displays a `Label`, a `TextField`, and two `Buttons` in a `GridPane`. Buttons are used to take the snapshot of the `TextField` synchronously and asynchronously. Click one of the `Buttons` to take a snapshot. A file save dialog appears for you to enter the file name for the saved snapshot.

The `syncSnapshot()` and `asyncSnapshot()` methods contain the logic to take the snapshot. For the snapshot, the fill is set to red, and a `Scale` and a `Rotate` transforms are applied. Figure 24-6 shows the snapshot.

#### ***Listing 24-10.*** Taking a Snapshot of a Node

```
// NodeSnapshot.java
package com.jdojo.image;

import javafx.application.Application;
import javafx.scene.Node;
import javafx.scene.Scene;
```

```

import javafx.scene.SnapshotParameters;
import javafx.scene.SnapshotResult;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.image.WritableImage;
import javafx.scene.layout.GridPane;
import javafx.scene.paint.Color;
import javafx.scene.transform.Rotate;
import javafx.scene.transform.Scale;
import javafx.scene.transform.Transform;
import javafx.stage.Stage;
import javafx.util.Callback;

public class NodeSnapshot extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        GridPane root = new GridPane();

        Label nameLbl = new Label("Name:");
        TextField nameField = new TextField("Prema");

        Button syncSnapshotBtn = new Button("Synchronous
Snapshot");
        syncSnapshotBtn.setOnAction(e ->
syncSnapshot(nameField));

        Button asyncSnapshotBtn = new Button("Asynchronous
Snapshot");
        asyncSnapshotBtn.setOnAction(e ->
asyncSnapshot(nameField));

        root.setHgap(10);
        root.addRow(0, nameLbl, nameField, syncSnapshotBtn);
        root.add(asyncSnapshotBtn, 2, 1);

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Taking the Snapshot of a Node");
        stage.show();
    }

    private void syncSnapshot(Node node) {
        SnapshotParameters params = getParams();
        WritableImage image = node.snapshot(params, null);
        ImageUtil.saveToFile(image);
    }

    private void asyncSnapshot(Node node) {
        // Create a Callback. Its call() method is called
when

```

```

        // the snapshot is ready. The getImage() method
returns the snapshot
        Callback<SnapshotResult, Void> callback
= (SnapshotResult result) -> {
            WritableImage image = result.getImage();
            ImageUtil.saveToFile(image);
            return null;
    };

    SnapshotParameters params = getParams();
    node.snapshot(callback, params, null);
}

private SnapshotParameters getParams() {
    // Set the fill to red and rotate the node by 30
degrees
    SnapshotParameters params = new SnapshotParameters();
    params.setFill(Color.RED);
    Transform tf = new Scale(0.8, 0.8);
    tf = tf.createConcatenation(new Rotate(10));
    params.setTransform(tf);
    return params;
}
}

```



**Figure 24-6.** The snapshot of a node

## Taking the Snapshot of a Scene

The `Scene` class contains an overloaded `snapshot()` method:

- `WritableImage snapshot(WritableImage image)`
- `void snapshot(Callback<SnapshotResult,Void> callback, WritableImage image)`

Compare the `snapshot()` methods of the `Scene` class with that of the `Node` class. The only difference is that the `snapshot()` method in the `Scene` class does not contain the `SnapshotParameters` argument. This means that you cannot customize the scene snapshot. Except this, the method works the same way as it works for the `Node` class, as discussed in the previous section.

The first version of the `snapshot()` method is synchronous whereas the second one is asynchronous. You can specify a `WritableImage` to the method that will hold the snapshot of the node. If this is `null`, a

`new WritableImage` is created. If the specified `WritableImage` is smaller than the scene, the scene will be clipped to fit the image size.

The program in Listing 24-11 shows how to take a snapshot of a scene. The main logic in the program is essentially the same as that of the program in Listing 24-10, except that, this time, it takes a snapshot of a scene. Figure 24-7 shows the snapshot.

### ***Listing 24-11.*** Taking a Snapshot of a Scene

```
// SceneSnapshot.java
package com.jdojo.image;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.image.WritableImage;
import javafx.scene.layout.GridPane;;
import javafx.scene.SnapshotResult;
import javafx.util.Callback;
import javafx.stage.Stage;

public class SceneSnapshot extends Application {
    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        GridPane root = new GridPane();
        Scene scene = new Scene(root);

        Label nameLbl = new Label("Name:");
        TextField nameField = new TextField("Prema");

        Button syncSnapshotBtn = new Button("Synchronous
Snapshot");
        syncSnapshotBtn.setOnAction(e ->
syncSnapshot(scene));

        Button asyncSnapshotBtn = new Button("Asynchronous
Snapshot");
        asyncSnapshotBtn.setOnAction(e ->
asyncSnapshot(scene));

        root.setHgap(10);
        root.addRow(0, nameLbl, nameField, syncSnapshotBtn);
        root.add(asyncSnapshotBtn, 2, 1);

        stage.setScene(scene);
        stage.setTitle("Taking the Snapshot of a Scene");
        stage.show();
    }
}
```

```

    }

    private void syncSnapshot(Scene scene) {
        WritableImage image = scene.snapshot(null);
        ImageUtil.saveToFile(image);
    }

    private void asyncSnapshot(Scene scene) {
        // Create a Callback. Its call() method is called
when
        // the snapshot is ready. The getImage() method
returns the snapshot
        Callback<SnapshotResult, Void> callback
= (SnapshotResult result) -> {
            WritableImage image = result.getImage();
            ImageUtil.saveToFile(image);
            return null;
        };
        scene.snapshot(callback, null);
    }
}

```

**Name:** Prema Synchronous Snapshot Asynchronous Snapshot

**Figure 24-7.** The snapshot of a scene

## Summary

JavaFX provides the Image API that lets you load and display images, and read/write raw image pixels. All classes in the API are in the `javafx.scene.image` package. The API lets you perform the following operations on images: load an image in memory, display an image as a node in a scene graph, read pixels from an image, write pixels to an image, and convert a node in a scene graph to an image and save it to the local file system.

An instance of the `Image` class is an in-memory representation of an image. You can also construct an image in a JavaFX application by supplying pixels to a `WritableImage` instance. The `Image` class supports BMP, PNG, JPEG, and GIF image formats. It loads an image from a source, which can be specified as a string URL or an `InputStream`. It can also scale the original image while loading.

An instance of the `ImageView` class is used to display an image loaded in an `Image` object. The `ImageView` class inherits from the `Node` class, which makes an `ImageView` suitable to be added to a scene graph.

Images are constructed from pixels. JavaFX supports reading pixels from an image, writing pixels to an image, and creating a snapshot of the scene. It supports creating an image from scratch. If an image is writable, you can also modify the image in memory and save it to the file system. The image API provides access to each pixel in the image. It supports reading and writing one pixel or a chunk of pixel at a time.

Data for pixels in an image may be stored in different formats. A `PixelFormat` defines how the data for a pixel for a given format is stored. A `WritablePixelFormat` represents a destination format to write pixels with full pixel color information.

The `PixelReader` and `PixelWriter` interfaces define methods to read data from an `Image` and write data to a `WritableImage`. Besides an `Image`, you can read pixels from and write pixels to any surface that contain pixels.

JavaFX allows you to take a snapshot of a `Node` and a `Scene` as they will appear in the next frame. You get the snapshot in a `WritableImage`, which means you can perform all pixel-level operations after you take the snapshot. The `Node` and `Scene` classes contain a `snapshot()` method to accomplish this.

The next chapter will discuss how to draw on a canvas using the Canvas API.

## CHAPTER 25



### Drawing on a Canvas

In this chapter, you will learn:

- What the Canvas API is
- How to create a canvas
- How to draw on a canvas such as basic shapes, text, paths, and images
- How to clear the canvas area
- How to save and restore the drawing states in a `GraphicsContext`

### What Is the Canvas API?

Through the `javafx.scene.canvas` package, JavaFX provides the Canvas API that offers a drawing surface to draw shapes, images, and text using drawing commands. The API also gives pixel-level access to the drawing surface where you can write any pixels on the surface. The API consists of only two classes:

- `Canvas`
- `GraphicsContext`

A canvas is a bitmap image, which is used as a drawing surface. An instance of the `Canvas` class represents a canvas. It inherits from the `Node` class. Therefore, a canvas is a node. It can be added to a scene graph, and effects and transformations can be applied to it.

A canvas has a graphics context associated with it that is used to issue drawing commands to the canvas. An instance of the `GraphicsContext` class represents a graphics context.

### Creating a Canvas

The `Canvas` class has two constructors. The no-args constructor creates an empty canvas. Later, you can set the size of the canvas using its `width` and `height` properties. The other constructor takes the width and height of the canvas as parameters:

```
// Create a Canvas of zero width and height  
Canvas canvas = new Canvas();
```

```
// Set the canvas size  
canvas.setWidth(400);  
canvas.setHeight(200);  
  
// Create a 400X200 canvas  
Canvas canvas = new Canvas(400, 200);
```

## Drawing on the Canvas

Once you create a canvas, you need to get its graphics context using the `getGraphicsContext2D()` method, as in the following snippet of code:

```
// Get the graphics context of the canvas  
GraphicsContext gc = canvas.getGraphicsContext2D();
```

All drawing commands are provided in the `GraphicsContext` class as methods. Drawings that fall outside the bounds of the canvas are clipped. The canvas uses a buffer. The drawing commands push necessary parameters to the buffer. It is important to note that you should use the graphics context from any one thread before adding the `Canvas` to the scene graph. Once the `Canvas` is added to the scene graph, the graphics context should be used only on the JavaFX Application Thread. The `GraphicsContext` class contains methods to draw the following types of objects:

- Basic shapes
- Text
- Paths
- Images
- Pixels

### Drawing Basic Shapes

The `GraphicsContext` class provides two types of methods to draw the basic shapes. The method `fillXXX()` draws a shape `XXX` and fills it with the current fill paint. The method `strokeXXX()` draws a shape `XXX` with the current stroke. Use the following methods for drawing shapes:

- `fillArc()`
- `fillOval()`
- `fillPolygon()`
- `fillRect()`
- `fillRoundRect()`
- `strokeArc()`
- `strokeLine()`

- `strokeOval()`
- `strokePolygon()`
- `strokePolyline()`
- `strokeRect()`
- `strokeRoundRect()`

The following snippet of code draws a rectangle. The stroke color is red and the stroke width is 2px. The upper-left corner of the rectangle is at (0, 0). The rectangle is 100px wide and 50px high.

```
Canvas canvas = new Canvas(200, 100);
GraphicsContext gc = canvas.getGraphicsContext2D();
gc.setLineWidth(2.0);
gc.setStroke(Color.RED);
gc.strokeRect(0, 0, 100, 50);
```

## Drawing Text

You can draw text using the `fillText()` and `strokeText()` methods of the `GraphicsContext` using the following snippets of code:

- `void strokeText(String text, double x, double y)`
- `void strokeText(String text, double x, double y, double maxWidth)`
- `void fillText(String text, double x, double y)`
- `void fillText(String text, double x, double y, double maxWidth)`

Both methods are overloaded. One version lets you specify the text and its position. The other version lets you specify the maximum width of the text as well. If the actual text width exceeds the specified maximum width, the text is resized to fit the specified the maximum width. The following snippet of code draws two strings. Figure 25-1 shows the two strings on the canvas.

```
Canvas canvas = new Canvas(200, 50);
GraphicsContext gc = canvas.getGraphicsContext2D();
gc.setLineWidth(1.0);
gc.setStroke(Color.BLACK);
gc.strokeText("Drawing Text", 10, 10);
gc.strokeText("Drawing Text", 100, 10, 40);
```

**Figure 25-1.** Drawing text on a canvas

## Drawing Paths

You can use path commands and SVG path strings to create a shape of your choice. A path consists of multiple subpaths. The following methods are used to draw paths:

- `beginPath()`
- `lineTo(double x1, double y1)`
- `moveTo(double x0, double y0)`
- `quadraticCurveTo(double xc, double yc, double x1, double y1)`
- `appendSVGPath(String svgpath)`
- `arc(double centerX, double centerY, double radiusX, double radiusY, double startAngle, double length)`
- `arcTo(double x1, double y1, double x2, double y2, double radius)`
- `bezierCurveTo(double x1, double y1, double x2, double y2, double x3, double y3)`
- `closePath()`
- `stroke()`
- `fill()`

The `beginPath()` and `closePath()` methods start and close a path, respectively. Methods such as `arcTo()` and `lineTo()` are the path commands to draw a specific type of subpath. Do not forget to call the `stroke()` or `fill()` method at the end, which will draw an outline or fill the path. The following snippet of code draws a triangle, as shown in Figure 25-2.

```
Canvas canvas = new Canvas(200, 50);
GraphicsContext gc = canvas.getGraphicsContext2D();
gc.setLineWidth(2.0);
gc.setStroke(Color.BLACK);

gc.beginPath();
gc.moveTo(25, 0);
gc.appendSVGPath("L50, 25L0, 25");
gc.closePath();
gc.stroke();
```



**Figure 25-2.** Drawing a triangle

## Drawing Images

You can draw an image on the canvas using the `drawImage()` method. The method has three versions:

- `void drawImage(Image img, double x, double y)`
- `void drawImage(Image img, double x, double y, double w, double h)`
- `void drawImage(Image img, double sx, double sy, double sw, double sh, double dx, double dy, double dw, double dh)`

You can draw the whole or part of the image. The drawn image can be stretched or shortened on the canvas. The following snippet of code draws the whole image in its original size on the canvas at (10, 10):

```
Image image = new Image("your_image_URL");
Canvas canvas = new Canvas(400, 400);
GraphicsContext gc = canvas.getGraphicsContext2D();
gc.drawImage(image, 10, 10);
```

The following statement will draw the whole image on the canvas by resizing it to fit in a 100px wide by 150px high area. Whether the image is stretched or shortened depends on its original size.

```
// Draw the whole image in 100X150 area at (10, 10)
gc.drawImage(image, 10, 10, 100, 150);
```

The following statement will draw part of an image on the canvas. Here it is assumed that the source image is bigger than 100px by 150px. The image part being drawn is 100px wide and 150px high and its upper left corner is at (0, 0) in the source image. The part of the image is drawn on the canvas at (10, 10) and it is stretched to fit 200px wide and 200px high area on the canvas.

```
// Draw part of the image in 200X200 area at (10, 10)
gc.drawImage(image, 0, 0, 100, 150, 10, 10, 200, 200);
```

## Writing Pixels

You can also directly modify pixels on the canvas.

The `getPixelWriter()` method of the `GraphicsContext` object returns a `PixelWriter` that can be used to write pixels to the associated canvas:

```
Canvas canvas = new Canvas(200, 100);
GraphicsContext gc = canvas.getGraphicsContext2D();
PixelWriter pw = gc.getPixelWriter();
```

Once you get a `PixelWriter`, you can write pixels to the canvas. Chapter 24 presented more details on how to write pixels using a `PixelWriter`.

## Clearing the Canvas Area

The canvas is a transparent area. Pixels will have colors and opacity depending on what is drawn at those pixels. Sometimes you may want to clear the whole or part of the canvas so the pixels are transparent again. The `clearRect()` method of the `GraphicsContext` lets you clear a specified area on the canvas:

```
// Clear the top-left 100X100 rectangular area from the canvas
gc.clearRect(0, 0, 100, 100);
```

## Saving and Restoring the Drawing States

The current settings for the `GraphicsContext` are used for all subsequent drawing. For example, if you set the line width to 5px, all subsequent strokes will be 5px in width. Sometimes you may want to modify the state of the graphics context temporarily, and after some time, restore the state that existed before the modification.

The `save()` and `restore()` methods of the `GraphicsContext` object let you save the current state and restore it afterward, respectively. Before you use these methods, let's discuss its need. Suppose you want to issue the following commands to the `GraphicsContext` object in order:

- Draw a rectangle without any effects
- Draw a string with a reflection effect
- Draw a rectangle without any effects

The following is the first (and incorrect) attempt of achieving this:

```
Canvas canvas = new Canvas(200, 120);
GraphicsContext gc = canvas.getGraphicsContext2D();
gc.strokeRect(10, 10, 50, 20);
gc.setEffect(new Reflection());
gc.strokeText("Chatar", 70, 20);
gc.strokeRect(120, 10, 50, 20);
```

Figure 25-3 shows the drawing of the canvas. Notice that the reflection effect was also applied to the second rectangle, which was not wanted.



**Figure 25-3.** Drawing shapes and text

You can fix the problem by setting the `Effect` to `null` after you draw the text. You had modified several properties for the `GraphicsContext` then had to restore them all manually.

Sometimes a `GraphicsContext` may be passed to your code but you do not want to modify its existing state.

The `save()` method stores the current state of the `GraphicsContext` on a stack. The `restore()` method restores the state of the `GraphicsContext` to the last saved state. Figure 25-4 shows the results of this. You can fix the problem using the following methods:

```
Canvas canvas = new Canvas(200, 120);
GraphicsContext gc = canvas.getGraphicsContext2D();

gc.strokeRect(10, 10, 50, 20);

// Save the current state
gc.save();

// Modify the current state to add an effect and draw the text
gc.setEffect(new Reflection());
gc.strokeText("Chatar", 70, 20);

// Restore the state what it was when the last save() was called
// and draw the second rectangle
gc.restore();
gc.strokeRect(120, 10, 50, 20);
```



**Figure 25-4.** Drawing shapes and text using `save()` and `restore()` methods

## A Canvas Drawing Example

The program in Listing 25-1 shows how to draw basic shapes, text, images, and raw pixels to a canvas. Figure 25-5 shows the resulting canvas with all drawings.

### **Listing 25-1.** Drawing on a Canvas

```
// CanvasTest.java
package com.jdojo.canvas;

import java.nio.ByteBuffer;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.image.Image;
import javafx.scene.image.PixelFormat;
import javafx.scene.image.PixelWriter;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
```

```
import javafx.stage.Stage;

public class CanvasTest extends Application {
    private static final int RECT_WIDTH = 20;
    private static final int RECT_HEIGHT = 20;

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

    @Override
    public void start(Stage stage) {
        Canvas canvas = new Canvas(400, 100);
        GraphicsContext gc = canvas.getGraphicsContext2D();

        // Set line width and fill color
        gc.setLineWidth(2.0);
        gc.setFill(Color.RED);

        // Draw a rounded rectangle
        gc.strokeRoundRect(10, 10, 50, 50, 10, 10);

        // Fill an oval
        gc.fillOval(70, 10, 50, 20);

        // Draw text
        gc.strokeText("Hello Canvas", 10, 85);

        // Draw an Image
        String imagePath = "resources/picture/ksharan.jpg";
        Image image = new Image(imagePath);
        gc.drawImage(image, 130, 10, 60, 80);

        // Write custom pixels to create a pattern
        writePixels(gc);

        Pane root = new Pane();
        root.getChildren().add(canvas);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Drawing on a Canvas");
        stage.show();
    }

    private void writePixels(GraphicsContext gc) {
        byte[] pixels = this.getBytesData();
        PixelWriter pixelWriter = gc.getPixelWriter();

        // Our data is in BYTE_RGB format
        PixelFormat<ByteBuffer> pixelFormat
        = PixelFormat.getByteRgbInstance();

        int spacing = 5;
        int imageWidth = 200;
        int imageHeight = 100;
```

```

// Roughly compute the number of rows and columns
int rows = imageHeight/(RECT_HEIGHT + spacing);
int columns = imageWidth/(RECT_WIDTH + spacing);

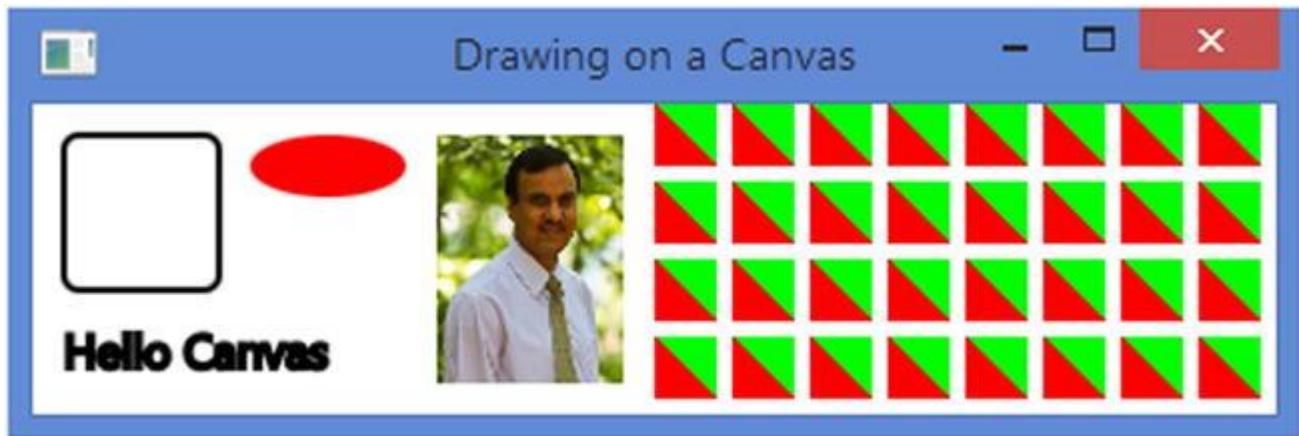
// Write the pixels to the canvas
for (int y = 0; y < rows; y++) {
    for (int x = 0; x < columns; x++) {
        int xPos = 200 + x * (RECT_WIDTH
+ spacing);
        int yPos = y * (RECT_HEIGHT + spacing);
        pixelWriter.setPixels(xPos, yPos,
                           RECT_WIDTH,
                           RECT_HEIGHT,
                           pixelFormat,
                           pixels, 0,
                           RECT_WIDTH * 3);
    }
}

private byte[] getPixelsData() {
    // Each pixel in the w X h region will take 3 bytes
    byte[] pixels = new byte[RECT_WIDTH * RECT_HEIGHT
* 3];

    // Height to width ration
    double ratio = 1.0 * RECT_HEIGHT/RECT_WIDTH;

    // Generate pixel data
    for (int y = 0; y < RECT_HEIGHT; y++) {
        for (int x = 0; x < RECT_WIDTH; x++) {
            int i = y * RECT_WIDTH * 3 + x * 3;
            if (x <= y/ratio) {
                pixels[i] = -1; // red -1 means
255 (-1 & 0xff = 255)
                pixels[i+1] = 0; // green = 0
                pixels[i+2] = 0; // blue = 0
            } else {
                pixels[i] = 0; // red = 0
                pixels[i+1] = -1; // Green 255
                pixels[i+2] = 0; // blue = 0
            }
        }
    }
    return pixels;
}
}

```



**Figure 25-5.** A canvas with shapes, text, images, and raw pixels on drawn on it

## Summary

Through the `javafx.scene.canvas` package, JavaFX provides the Canvas API that offers a drawing surface to draw shapes, images, and text using drawing commands. The API also gives pixel-level access to the drawing surface where you can write any pixels on the surface. The API consists of only two classes: `Canvas` and `GraphicsContext`. A canvas is a bitmap image, which is used as a drawing surface. An instance of the `Canvas` class represents a canvas. It inherits from the `Node` class. Therefore, a canvas is a node. It can be added to a scene graph, and effects and transformations can be applied to it. A canvas has a graphics context associated with it that is used to issue drawing commands to the canvas. An instance of the `GraphicsContext` class represents a graphics context.

The `Canvas` class contains a `getGraphicsContext2D()` method that returns an instance of the `GraphicsContext` class. After obtaining the `GraphicsContext` of a canvas, you issue drawing commands to the `GraphicsContext` that performs the drawing.

Drawings falling outside the bounds of the canvas are clipped. The canvas uses a buffer. The drawing commands push necessary parameters to the buffer. The `GraphicsContext` of a canvas can be used from any one thread before the canvas is added to the scene graph. Once the canvas is added to the scene graph, the graphics context should be used only on the JavaFX Application Thread. The `GraphicsContext` class contains methods to draw the following types of objects: basic shapes, text, paths, images, and pixels.

The next chapter will discuss how to use the drag-and-drop gesture to transfer data between nodes in the same JavaFX application, between two different JavaFX applications, and between a JavaFX application and a native application.

## CHAPTER 26



### Understanding Drag and Drop

In this chapter, you will learn:

- What a press-drag-release gesture is
- How to use a dragboard to facilitate data transfers
- How to initiate and detect a drag-and-drop gesture
- How to transfer data from the source to the target using a drag-and-drop gesture
- How to transfer images using a drag-and-drop gesture
- How to transfer custom data between the source and the target using a drag-and-drop gesture

#### What Is a Press-Drag-Release Gesture?

A press-drag-release gesture is a user action of pressing a mouse button, dragging the mouse with the pressed button, and releasing the button. The gesture can be initiated on a scene or a node. Several nodes and scenes may participate in a single press-drag-release gesture. The gesture is capable of generating different types of events and delivering those events to different nodes. The type of generated events and nodes receiving the events depends on the purpose of the gesture. A node can be dragged for different purposes:

- You may want to change the shape of a node by dragging its boundaries or move it by dragging it to a new location. In this case, the gesture involves only one node: the node on which the gesture was initiated.
- You may want to drag a node and drop it onto another node to connect them in some fashion, for example, connecting two nodes with a symbol in a flow chart. In this case, the drag gesture involves multiple nodes. When the source node is dropped onto the target node, an action takes place.
- You can drag a node and drop it onto another node to transfer data from the source node to the target node. In this case, the drag gesture involves multiple nodes. A data transfer occurs when the source node is dropped.

JavaFX supports three types of drag gestures:

- A simple press-drag-release gesture
- A full press-drag-release gesture
- A drag-and-drop gesture

This chapter will focus mainly on the third type of gesture: the drag-and-drop gesture. It is essential to understand the first two types of gestures to gain full insight into the drag-and-drop gesture. I will discuss the first two types of gestures briefly with a simple example of each type.

### A Simple Press-Drag-Release Gesture

The *simple press-drag-release* gesture is the default drag gesture. It is used when the drag gesture involves only one node—the node on which the gesture was initiated. During the drag gesture, all `MouseEvent` types—mouse-drag entered, mouse-drag over, mouse-drag exited, mouse, and mouse-drag released—are delivered only to the gesture source node. In this case, when the mouse button is pressed, the topmost node is picked and all subsequent mouse events are delivered to that node until the mouse button is released. When the mouse is dragged onto another node, the node on which the gesture was started is still under the cursor and, therefore, no other nodes receive the events until the mouse button is released.

The program in Listing 26-1 demonstrates a case of the simple press-drag-release gesture. It adds two `TextFields` to a scene: one is called the source node and the other the target node. Event handlers are added to both nodes. The target node adds `MouseEvent` handlers to detect any mouse-drag event on it. Run the program, press the mouse button on the source node, drag it onto the target node, and, finally, release the mouse button. The output that follows shows that the source node receives all mouse-drag events. The target node does not receive any mouse-drag events. This is the case of a simple press-drag-release gesture where the node initiating the drag gesture receives all mouse-drag events.

#### ***Listing 26-1.*** Demonstrating a Simple Press-Drag-Release Gesture

```
// SimplePressDragRelease.java
package com.jdojo.dnd;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;
```

```

public class SimplePressDragRelease extends Application {
    TextField sourceFld = new TextField("Source Node");
    TextField targetFld = new TextField("Target node");

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

    @Override
    public void start(Stage stage) {
        // Build the UI
        GridPane root = getUI();

        // Add event handlers
        this.addEventHandlers();

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("A simple press-drag-release
gesture");
        stage.show();
    }

    private GridPane getUI() {
        GridPane pane = new GridPane();
        pane.setHgap(5);
        pane.setVgap(20);
        pane.addRow(0, new Label("Source Node:"), sourceFld);
        pane.addRow(1, new Label("Target Node:"), targetFld);
        return pane;
    }

    private void addEventHandlers() {
        // Add mouse event handlers for the source
        sourceFld.setOnMousePressed(e -> print("Source:
pressed"));
        sourceFld.setOnMouseDragged(e -> print("Source:
dragged"));
        sourceFld.setOnDragDetected(e -> print("Source:
dragged detected"));
        sourceFld.setOnMouseReleased(e -> print("Source:
released"));

        // Add mouse event handlers for the target
        targetFld.setOnMouseDragEntered(e -> print("Target:
drag entered"));
        targetFld.setOnMouseDragOver(e -> print("Target:
drag over"));
        targetFld.setOnMouseDragReleased(e -> print("Target:
drag released"));
        targetFld.setOnMouseDragExited(e -> print("Target:
drag exited"));
    }
}

```

```
    private void print(String msg) {
        System.out.println(msg);
    }
}

Source: Mouse pressed
Source: Mouse dragged
Source: Mouse dragged detected
Source: Mouse dragged
Source: Mouse dragged

...
Source: Mouse released
```

Note that the drag-detected event is generated once after the mouse is dragged. The `MouseEvent` object has a `dragDetect` flag, which can be set in the mouse-pressed and mouse-dragged events. If it is set to true, the subsequent event that is generated is the drag-detected event. The default is to generate it after the mouse-dragged event. If you want to generate it after the mouse-pressed event, not the mouse-dragged event, you need to modify the event handlers:

```
sourceFld.setOnMousePressed(e -> {
    print("Source: Mouse pressed");

    // Generate drag detect event after the current mouse
    // pressed event
    e.setDragDetect(true);
});

sourceFld.setOnMouseDragged(e -> {
    print("Source: Mouse dragged");

    // Suppress the drag detected default event generation
    // after mouse dragged
    e.setDragDetect(false);
});
```

## A Full Press-Drag-Release Gesture

When the source node of a drag gesture receives the drag-detected event, you can start a *full press-drag-release* gesture by calling the `startFullDrag()` method on the source node.

The `startFullDrag()` method exists in both `Node` and `Scene` classes, allowing you to start a full press-drag-release gesture for a node and a scene. I will simply use only the term `node` during this discussion.

**Tip** The `startFullDrag()` method can only be called from the drag-detected event handler. Calling this method from any other place throws an `IllegalStateException`.

You need to do one more set up to see the full press-drag-release gesture in action. The source node of the drag gesture will still receive all mouse-drag events as it is under the cursor when a drag is happening. You need to set the `mouseTransparent` property of the gesture source to false so the node below it will be picked and mouse-drag events will be delivered to that node. Set this property to true in the mouse-pressed event and set it back to false in the mouse-released event.

The program in Listing 26-2 demonstrates a full press-drag-release gesture. The program is similar to the one show in Listing 26-1, except for the following:

- In the mouse-pressed event handler for the source node, the `mouseTransparent` property for the source node is set to false. It is set back to true in the mouse-released event handler.
- In the drag-detected event handler, the `startFullDrag()` method is called on the source node.

Run the program, press the mouse button on the source node, drag it onto the target node, and, finally, release the mouse button. The output that follows shows that the target node receives mouse-drag events as the mouse is dragged inside its bounds. This is the case of a full press-drag-release gesture where the node over which the mouse drag takes place receives the mouse-drag events.

### ***Listing 26-2.*** Demonstrating a Full Press-Drag-Release Gesture

```
// FullPressDragRelease.java
package com.jdojo.dnd;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class FullPressDragRelease extends Application {
    TextField sourceFld = new TextField("Source Node");
    TextField targetFld = new TextField("Target node");

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

    @Override
    public void start(Stage stage) {
        // Build the UI
    }
}
```

```

        GridPane root = getUI();

        // Add event handlers
        this.addEventHanders();

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("A full press-drag-release gesture");
        stage.show();
    }

    private GridPane getUI() {
        GridPane pane = new GridPane();
        pane.setHgap(5);
        pane.setVgap(20);
        pane.addRow(0, new Label("Source Node:"), sourceFld);
        pane.addRow(1, new Label("Target Node:"), targetFld);
        return pane;
    }

    private void addEventHanders() {
        // Add mouse event handlers for the source
        sourceFld.setOnMousePressed(e -> {
            // Make sure the node is not picked
            sourceFld.setMouseTransparent(true);
            print("Source: Mouse pressed");
        });

        sourceFld.setOnMouseDragged(e -> print("Source: Mouse dragged"));

        sourceFld.setOnDragDetected(e -> {
            // Start a full press-drag-release gesture
            sourceFld.startFullDrag();
            print("Source: Mouse dragged detected");
        });

        sourceFld.setOnMouseReleased(e -> {
            // Make sure the node is picked
            sourceFld.setMouseTransparent(false);
            print("Source: Mouse released");
        });

        // Add mouse event handlers for the target
        targetFld.setOnMouseDragEntered(e -> print("Target: drag entered"));
        targetFld.setOnMouseDragOver(e -> print("Target: drag over"));
        targetFld.setOnMouseDragReleased(e -> print("Target: drag released"));
        targetFld.setOnMouseDragExited(e -> print("Target: drag exited"));
    }
}

```

```
        private void print(String msg) {
            System.out.println(msg);
        }
    }
Source: Mouse pressed
Source: Mouse dragged
Source: Mouse dragged
Source: Mouse dragged detected
Source: Mouse dragged
Source: Mouse dragged
Target: drag entered
Target: drag over
Source: Mouse dragged
Target: drag over
Target: drag released
Source: Mouse released
Target: drag exited
```

## A Drag-and-Drop Gesture

The third type of drag gesture is called a *drag-and-drop* gesture, which is a user action combining the mouse movement with a pressed mouse button. It is used to transfer data from the *gesture source* to a *gesture target*. A drag-and-drop gesture allows transferring data from:

- One node to another node
- A node to a scene
- One scene to another scene
- A scene to a node

The source and target can be in the same Java or JavaFX application or two different Java or JavaFX applications. A JavaFX application and a native application may also participate in the gesture, for example:

- You can drag text from a Microsoft Word application to a JavaFX application to populate a `TextArea` and vice versa.
- You can drag an image file from Windows Explorer and drop it onto an `ImageView` in a JavaFX application.  
The `ImageView` can display the image.
- You can drag a text file from Windows Explorer and drop it onto a `TextArea` in a JavaFX application.  
The `TextArea` will read the file and display its content.

Several steps are involved in performing a drag-and-drop gesture:

- A mouse button is pressed on a node.
- The mouse is dragged with the button pressed.

- The node receives a drag-detected event.
- A drag-and-drop gesture is started on the node by calling the `startDragAndDrop()` method, making the node a the gesture source. The data from the source node is placed in a dragboard.
- Once the system switches to a drag-and-drop gesture, it stops delivering `MouseEvents` and starts delivering `DragEvents`.
- The gesture source is dragged onto the potential gesture target. The potential gesture target checks whether it accepts the data placed in the dragboard. If it accepts the data, it may become the actual gesture target. The node indicates whether it accepts the data in one of its `DragEvent` handlers.
- The user releases the pressed button on the gesture target, sending it a drag-dropped event.
- The gesture target uses the data from the dragboard.
- A drag-done event is sent to the gesture source indicating that the drag-and-drop gesture is complete.

I will discuss all of these steps in detail in the sections that follow. The classes supporting the drag-and-drop gesture are included in the `javafx.scene.input` package.

## Understanding the Data Transfer Modes

In a drag-and-drop gesture, the data can be transferred in three modes:

- Copy
- Move
- Link

The *copy* mode indicates that the data will be copied from the gesture source to the gesture target. You may drag a `TextField` and drop it onto another `TextField`. The latter gets a copy of the text contained in the former.

The *move* mode indicates that the data will be moved from the gesture source to the gesture target. You may drag a `TextField` and drop it onto another `TextField`. The text in the former is then moved to the latter.

The *link* mode indicates that the gesture target will create a link (or reference) to the data being transferred. The actual meaning of “link” depends on the application. You may drag and drop a URL to a `WebView` in the link mode. The `WebView` then loads the URL content.

The three data transfer modes are represented by the following three constants in the `TransferMode` enum:

- `TransferMode.COPY`
- `TransferMode.MOVE`
- `TransferMode.LINK`

Sometimes you may need a combination of the three transfer modes. The `TransferMode` enum contains three convenience static fields that are arrays of its enum constants:

- `TransferMode[] ANY`
- `TransferMode[] COPY_OR_MOVE`
- `TransferMode[] NONE`

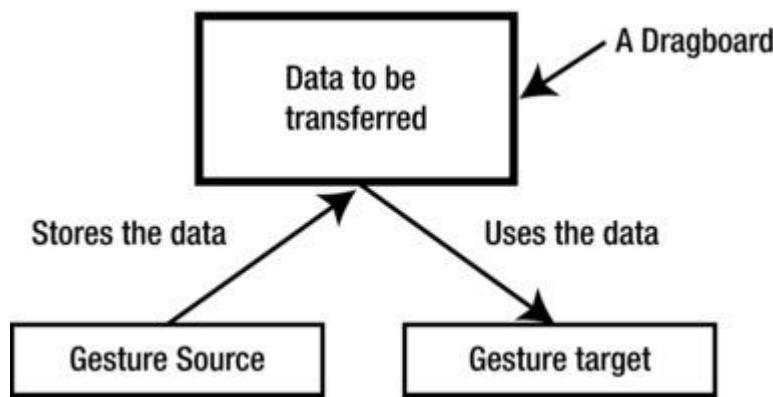
The `ANY` field is an array of `COPY`, `MOVE`, and `LINK` enum constants. The `COPY_OR_MOVE` field is an array of the `COPY` and `MOVE` enum constants. The `NONE` constant is an empty array.

Every drag-and-drop gesture includes the use of the `TransferMode` enum constants. The gesture source specifies the transfer modes that it supports for the data transfer. The gesture target specifies the modes in which it accepts the data transfer.

## **Understanding the *Dragboard***

In a drag-and-drop data transfer, the gesture source and the gesture target do not know each other. In fact, they may belong to two different applications: two JavaFX applications, or one JavaFX and one native. How does the data transfer take place between the gesture source and target if they do not know each other? In the real world, an intermediary is needed to facilitate a transaction between two unknown parties. In a drag-and-drop gesture, an intermediary is also used to facilitate the data transfer.

A dragboard acts as an intermediary between the gesture source and gesture target. A dragboard is the storage device that holds the data being transferred. The gesture source places the data into a dragboard; the dragboard is made available to the gesture target, so it can inspect the type of content that is available for transfer. When the gesture target is ready to transfer the data, it gets the data from the dragboard. Figure 26-1 shows the roles played by a dragboard.



**Figure 26-1.** Data transfer mechanism in a drag-and-drop gesture

An instance of the `Dragboard` class represents a dragboard. The class is inherited from the `Clipboard` class. An instance of the `Clipboard` class represents an operating system clipboard. Typically, an operating system uses a clipboard to store data during cut, copy, and paste operations. You can get the reference of the general clipboard of the operating system using the

`static getSystemServiceClipboard()` method of the `Clipboard` class:

```
Clipboard systemClipboard = Clipboard.getSystemServiceClipboard();
```

You can place data in the system clipboard that will be accessible to all applications in the system. You can read the data placed in the system clipboard, which can be placed there by any application. A clipboard can store different types of data, for example, rich text format (RTF) text, plain text, HTML, URL, images, or files. The class contains several methods to check if data in a specific format are available in the clipboard. These methods return `true` if the data in the specific format are available. For example, the `hasString()` method returns `true` if the clipboard contains a plain string; the `hasRtf()` method returns `true` for text in rich text format. The class contains methods to retrieve data in the specific format. For example, the `getString()` method returns data in plain text format; the `getHtml()` returns HTML text; the `getImage()` returns an image, and so forth. The `clear()` method clears the clipboard.

**Tip** You cannot create an instance of the `Clipboard` class directly. The clipboard is meant to store one *conceptual* item. The term conceptual means that the data in the clipboard may be stored in different formats representing the same item. For example, you may store RTF text and its plain text version. In this case, the clipboard has two copies of the same item in different formats.

The clipboard is not limited to store only a fixed number of data types. Any serializable data can be stored on the clipboard. Data stored on the clipboard has an associated data format. An instance of

the `DataFormat` class represents a data format. The `DataFormat` class contains six static fields to represent the commonly used data formats:

- FILES
- HTML
- IMAGE
- PLAIN\_TEXT
- RTF
- URL

The `FILES` represents a list of `java.io.File` objects.

The `HTML` represents an HTML-formatted string. The `IMAGE` represents a platform-specific image type. The `PLAIN_TEXT` represents a plain text string. The `RTF` represents an RTF-formatted string. The `URL` represents a URL encoded as a string.

You may want to store data in the clipboard in a format other than those listed above. You can create a `DataFormat` object to represent any arbitrary format. You need to specify a list of mime types for your data format. The following statement creates

a `DataFormat` with `jdojo/person` and `jdojo/personlist` as the mime types:

```
DataFormat myFormat = new DataFormat("jdojo/person",
"jdojo/person");
```

The `Clipboard` class provides the following methods to work with the data and its format:

- boolean `setContent(Map<DataFormat, Object> content)`
- Object `getContent(DataFormat dataFormat)`

The content of the clipboard is a map with the `DataFormat` as keys and data as values. The `getContent()` method returns null if data in the specific data format are not available in the clipboard. The following snippet of code stores HTML and plain text version of data, and later, retrieves the data in both formats:

```
// Store text in HTML and plain-text formats in the system
clipboard
Clipboard clipboard = Clipboard.getSystemClipboard();

Map<DataFormat, Object> data = new HashMap<>();
data.put(DataFormat.HTML, "<b>Yahoo!</b>");
data.put(DataFormat.PLAIN_TEXT, "Yahoo!");
clipboard.setContent(data);
...

// Try reading HTML text and plain text from the clipboard
```

```

If (clipboard.hasHtml()) {
    String htmlText
= (String)clipboard.getContent(DataFormat.HTML);
    System.out.println(htmlText);
}

If (clipboard.hasString()) {
    String plainText
= (String)clipboard.getContent(DataFormat.PLAIN_TEXT);
    System.out.println(plainText);
}

```

Preparing data to store in the clipboard requires writing a little bloated code. An instance of the `ClipboardContent` class represents the content of the clipboard, and it makes working with the clipboard data a little easier. The class inherits from the `HashMap<DataFormat, Object>` class. It provides convenience methods in the form `putXXX()` and `getXXX()` for commonly used data types. The following snippet of code rewrites the above logic to store data into the clipboard. The logic to retrieve the data remains the same.

```

Clipboard clipboard = Clipboard.getSystemClipboard();
ClipboardContent content = new ClipboardContent();
content.putHtml("<b>Yahoo!</b>");
content.putString("Yahoo!");
clipboard.setContent(content);

```

The `Dragboard` class contains all the methods available in the `Clipboard` class. It adds the following methods:

- `Set<TransferMode> getTransferModes()`
- `void setDragView(Image image)`
- `void setDragView(Image image, double offsetX, double offsetY)`
- `void setDragViewOffsetX(double offsetX)`
- `void setDragViewOffsetY(double offsetY)`
- `Image getDragView()`
- `Double getDragViewOffsetX()`
- `double getDragViewOffsetY()`

The `getTransferModes()` method returns the set of transfer modes supported by the gesture target. The `setDragView()` method sets an image as the drag view. The image is shown when the gesture source is dragged. The offsets are the x and y positions of the cursor over the image. Other methods involve getting the drag-view image and the cursor offsets.

**Tip** A dragboard is a special system clipboard used for the drag-and-drop gesture. You cannot create a dragboard explicitly. Whenever it is necessary to work with the dragboard, its reference is made available as the returned value

from methods or the property of the event object. For example, the `DragEvent` class contains a `getDragboard()` method that returns the reference of the `Dragboard` containing the data being transferred.

## The Example Application

In the following sections I will discuss the steps in a drag-and-drop gesture in detail, and you will build an example application. The application will have two `TextFields` displayed in a scene. One text field is called the source node and the other the target node. The user can drag and drop the source node over to the target node. Upon completion of the gesture, the text from source node is transferred (copied or moved) to the target node. I will refer to those nodes in the discussion. They are declared as follows:

```
TextField sourceFld = new TextField("Source node");
TextField targetFld = new TextField("Target node");
```

### Initiating the Drag-and-Drop Gesture

The first step in a drag-and-drop gesture is to convert a simple press-drag-release gesture into a drag-and-drop gesture. This is accomplished in the mouse-drag detected event handler for the gesture source. Calling the `startDragAndDrop()` method on the gesture source initiates a drag-and-drop gesture. The method is available in the `Node` and `Scene` classes, so a node and a scene can be the gesture source of a drag-and-drop gesture. The method signature is:

```
Dragboard startDragAndDrop(TransferMode... transferModes)
```

The method accepts the list of supported transfer modes by the gesture source and returns a dragboard. The gesture source needs to populate the dragboard with the data it intends to transfer. The following snippet of code initiates a drag-and-drop gesture, copies the source `TextField` text to the dragboard, and consumes the event. The drag-and-drop gesture is initiated only when the `TextField` contains text.

```
sourceFld.setOnDragDetected((MouseEvent e) -> {
    // User can drag only when there is text in the source
    // field
    String sourceText = sourceFld.getText();
    if (sourceText == null || sourceText.trim().equals("")) {
        e.consume();
        return;
    }

    // Initiate a drag-and-drop gesture
    Dragboard dragboard
    = sourceFld.startDragAndDrop(TransferMode.COPY_OR_MOVE);
```

```
// Add the source text to the Dragboard  
ClipboardContent content = new ClipboardContent();  
content.putString(sourceText);  
dragboard.setContent(content);  
  
e.consume();  
});
```

## Detecting a Drag Gesture

Once the drag-and-drop gesture has been initiated, you can drag the gesture source over to any other node. The gesture source has already put the data in the dragboard declaring the transfer modes that it supports. It is now time for the potential gesture targets to declare whether they accept the data transfer offered by the gesture source. Note that there could be multiple potential gesture targets. One of them will become the actual gesture target when the gesture source is dropped on it.

The potential gesture target receives several types of drag events:

- It receives a drag-entered event when the gesture source enters its bounds.
- It receives a drag-over event when the gesture source is dragged around within its bounds.
- It receives a drag-exited event when the gesture source exits its bounds.
- It receives a drag-dropped event when the gesture source is dropped over it by releasing the mouse button.

In a drag-over event handler, the potential gesture target needs to declare that it intends to participate in the drag-and-drop gesture by calling the `acceptTransferModes(TransferMode... modes)` method of the `DragEvent`. Typically, the potential target checks the content of the dragboard before declaring whether it accepts the transfer modes. The following snippet of code accomplishes this. The target `TextField` checks the dragboard for plain text. It contains plain text, so the target declares that it accepts `COPY` and `MOVE` transfer modes.

```
targetFld.setOnDragOver((DragEvent e) -> {  
    // If drag board has a string, let the event know that the  
    target accepts  
    // copy and move transfer modes  
    Dragboard dragboard = e.getDragboard();  
  
    if(dragboard.hasString()) {  
        e.acceptTransferModes(TransferMode.COPY_OR_MOVE);  
    }  
}
```

```

        e.consume();
    });
}

```

## Dropping the Source onto the Target

If the potential gesture target accepts the transfer mode supported by the gesture source, the gesture source can be dropped on the target. The dropping is accomplished by releasing the mouse button while the gesture source is still over the target. When the gesture source is dropped onto a target, the target becomes the actual gesture target. The actual gesture target receives the drag-dropped event. You need to add a drag-drop event handler for the gesture target in which it performs two tasks:

- It accesses the data in the dragboard.
- It calls the `setDropCompleted(boolean isTransferDone)` method of the `DragEvent` object.

Passing true to the method indicates that the data transfer was successful. Passing false indicates that the data transfer was unsuccessful. The dragboard cannot be accessed after calling this method.

The following snippet of code performs the data transfer and sets the appropriate completion flag:

```

targetFld.setOnDragDropped( DragEvent e ) -> {
    // Transfer the data to the target
    Dragboard dragboard = e.getDragboard();
    if(dragboard.hasString()) {
        String text = dragboard.getString();
        targetFld.setText(text);

        // Data transfer is successful
        e.setDropCompleted(true);
    } else {
        // Data transfer is not successful
        e.setDropCompleted(false);
    }

    e.consume();
};

```

## Completing the Drag-and-Drop Gesture

After the gesture source has been dropped, it receives a drag-done event. The `DragEvent` object contains a `getTransferMode()` method. When it is called from the drag-done event handler, it returns the transfer mode used for the data transfer. Depending on the transfer mode, you can clear or keep the content of the gesture source. For example, if the

transfer mode is MOVE, it is better to clear the source content to give the user a real feel of the data move.

You may wonder what determines the data transfer mode. In this example, both the gesture source and the target support COPY and MOVE. When the target accessed the data from the dragboard in the drag-dropped event, it did not set any transfer mode. The system determines the data transfer mode depending on the state of certain keys and the source and target. For example, when you drag a `TextField` and drop it onto another `TextField`, the default data transfer mode is MOVE. When the same drag and drop is performed with the Ctrl key pressed, the COPY mode is used.

If

the `getTransferMode()` method returns null or `TransferMode.ON_E`, it indicates that no data transfer happened. The following snippet of code handles the drag-done event for the source `TextField`. The source text is cleared if the data transfer mode was MOVE.

```
sourceFld.setOnDragDone( DragEvent e ) -> {
    // Check how the data transfer happened. If it was moved,
    // clear the text in the source.
    TransferMode modeUsed = e.getTransferMode();

    if ( modeUsed == TransferMode.MOVE ) {
        sourceFld.setText("");
    }

    e.consume();
} );
```

This completes handling of a drag-and-drop gesture. If you need more information about the parties participating in the drag-and-drop gesture, please refer to the API documentation for the `DragEvent` class. For example, use

the `getGestureSource()` and `getGestureTarget()` methods to get the reference of the gesture source and target, respectively.

## Providing Visual Clues

There are several ways to provide visual clues during a drag-and-drop gesture:

- The system provides an icon under the cursor during the drag gesture. The icon changes depending on the transfer mode determined by the system and whether the drag target is a potential target for the drag-and-drop gesture.
- You can write code for the drag-enter and drag-exited events for the potential targets by changing its visual appearance.

For example, in the drag-entered event handler, you can change the background color of the potential target to green if it allows the data transfer and to red if it does not. In the drag-exited event handler, you can change the background color back to normal.

- You can set a drag view in the dragboard in the drag-detected event handler for the gesture. The drag view is an image. For example, you can take a snapshot of the node or part of the node being dragged and set it as the drag view.

## A Complete Drag-and-Drop Example

The program in Listing 26-3 has the complete source code for this example. It displays a window as shown in Figure 26-2. You can drag the gesture source `TextField` and drop it onto the target `TextField`. The text from the source will be copied or moved to the target. The transfer mode depends on the system. For example, on Windows, pressing the `Ctrl` key while dropping will copy the text, and dropping without pressing the `Ctrl` key will move the text. Notice that the drag icon is changed during the drag action. The icon gives you a clue as to what kind of data transfer is going to happen when you drop the source. For example, when you drag the source on a target that does not accept the data transfer offered by the source, a “not-allowed” icon, a circle with a diagonal solid line, is displayed.

### ***Listing 26-3.*** Performing a Drag-and-Drop Gesture

```
// DragAndDropTest.java
package com.jdojo.dnd;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.input.ClipboardContent;
import javafx.scene.input.DragEvent;
import javafx.scene.input.Dragboard;
import javafx.scene.input.MouseEvent;
import javafx.scene.input.TransferMode;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class DragAndDropTest extends Application {
    TextField sourceFld = new TextField("JavaFX");
    TextField targetFld = new TextField("Drag and drop the
source text here");

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

```

        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // Build UI
        GridPane root = getUIs();

        // Add event handlers for the source and target
        this.addDnDEventHandlers();

        root.setStyle("-fx-padding: 10;" +
                      "-fx-border-style: solid inside;" +
                      "-fx-border-width: 2;" +
                      "-fx-border-insets: 5;" +
                      "-fx-border-radius: 5;" +
                      "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Performing a Drag-and-Drop
Gesture");
        stage.show();
    }

    private GridPane getUIs() {
        // Set prompt text
        sourceFld.setPromptText("Enter text to drag");
        targetFld.setPromptText("Drag the source text
here");

        GridPane pane = new GridPane();
        pane.setHgap(5);
        pane.setVgap(20);
        pane.add(new Label("Drag and drop the source text
field" +
                           " onto the target text field."), 0, 0,
2, 1);
        pane.addRow(1, new Label("DnD Gesture Source:"), sourceFld);
        pane.addRow(2, new Label("DnD Gesture Target:"), targetFld);
        return pane;
    }

    private void addDnDEventHandlers() {
        sourceFld.setOnDragDetected(this::dragDetected);
        targetFld.setOnDragOver(this::dragOver);
        targetFld.setOnDragDropped(this::dragDropped);
        sourceFld.setOnDragDone(this::dragDone);
    }

    private void dragDetected(MouseEvent e) {
        // User can drag only when there is text in the
source field
        String sourceText = sourceFld.getText();
    }
}

```

```

        if (sourceText == null || sourceText.trim().equals("")) {
            e.consume();
            return;
        }

        // Initiate a drag-and-drop gesture
        Dragboard dragboard =
            sourceFld.startDragAndDrop(TransferMode.COPY_OR_MOVE);

        // Add the source text to the Dragboard
        ClipboardContent content = new ClipboardContent();
        content.putString(sourceText);
        dragboard.setContent(content);

        e.consume();
    }

    private void dragOver(DragEvent e) {
        // If drag board has a string, let the event know
        // that the target accepts copy and move transfer modes
        Dragboard dragboard = e.getDragboard();
        if (dragboard.hasString()) {
            e.acceptTransferModes(TransferMode.COPY_OR_MOVE);
        }

        e.consume();
    }

    private void dragDropped(DragEvent e) {
        // Transfer the data to the target
        Dragboard dragboard = e.getDragboard();
        if (dragboard.hasString()) {
            String text = dragboard.getString();
            targetFld.setText(text);

            // Data transfer is successful
            e.setDropCompleted(true);
        } else {
            // Data transfer is not successful
            e.setDropCompleted(false);
        }

        e.consume();
    }

    private void dragDone(DragEvent e) {
        // Check how data was transferred to the target. If
        it was moved, clear the
        // text in the source.
        TransferMode modeUsed = e.getTransferMode();
    }
}

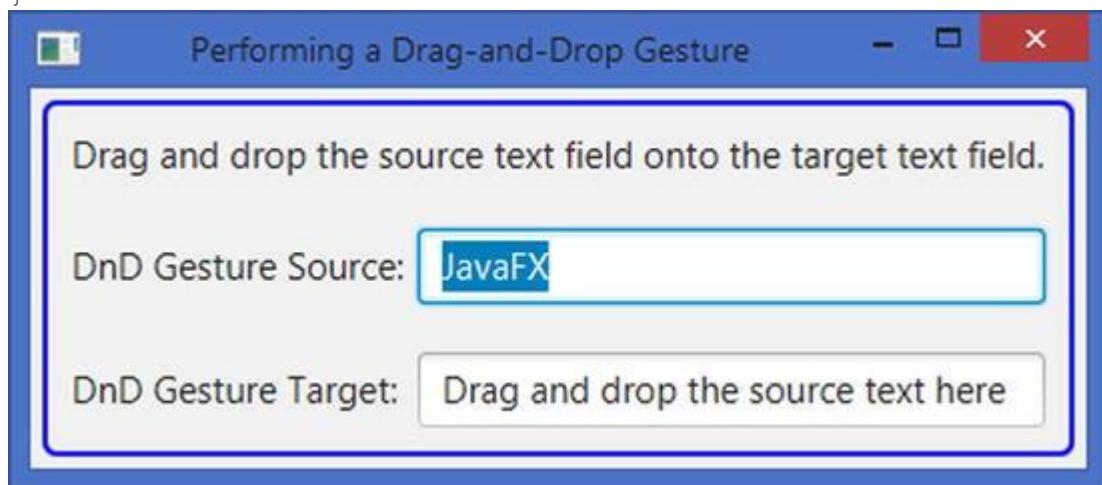
```

```

        if (modeUsed == TransferMode.MOVE) {
            sourceFld.setText("");
        }

        e.consume();
    }
}

```



**Figure 26-2.** A scene letting transfer text from a `TextField` to another using a drag-and-drop gesture

## Transferring an Image

The drag-and-drop gesture allows you to transfer an image. The image can be placed on the dragboard. You can also place a URL or a file on the dragboard that refers to the image location. Let's develop a simple application to demonstrate an image data transfer. To transfer an image, the user can drag and drop the following to a scene:

- An image
- An image file
- A URL pointing to an image

The program in Listing 26-4 opens a window with a text message, an empty `ImageView`, and a button. The `ImageView` will display the dragged and dropped image. Use the button to clear the image.

The entire scene is a potential target for a drag-and-drop gesture. A drag-over event handler is set for the scene. It checks whether the dragboard contains an image, a list of files, or a URL. If it finds one of these data types in the dragboard, it reports that it will accept ANY data transfer mode. In the drag-dropped event handler for the scene, the program attempts to read the image data, list of files, and the URL in order. If it is a list of files, you look at the mime type of each file to see if the name starts with `image/`. You use the first file with an image mime

type and ignore the rest. If it is a URL, you simply try creating an `Image` object from it. You can play with the application in different ways:

- Run the program and open the HTML file `drag_and_drop.html` in a browser. The file is included in the `src/resources/html` directory. The HTML file contains two links: one pointing to a local image file and the other to a remote image file. Drag and drop the links onto the scene. The scene will show the images referred to by the links. Drag and drop the image from the web page. The scene will display the image. (Dragging and dropping of the image worked fine in Mozilla and Google Chrome browsers, but not in Windows Explorer.)
- Open a file explorer, for example, Windows Explorer on Windows. Select an image file and drag and drop the file onto the scene. The scene will display the image from the file. You can drop multiple files, but the scene will display only an image from one of those files.

You can enhance the application by allowing the user to drag multiple files onto the scene and showing them all in a `TilePane`. You can also add more error checks and feedbacks to the user about the drag-and-drop gesture.

#### ***Listing 26-4.*** Transferring an Image Using a Drag-and-Drop Gesture

```
// ImageDragAndDrop.java
package com.jdojo.dnd;

import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.util.List;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.input.DragEvent;
import javafx.scene.input.Dragboard;
import javafx.scene.input.TransferMode;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ImageDragAndDrop extends Application {
    ImageView imageView = new ImageView();
```

```

Button clearBtn = new Button("Clear Image");
Scene scene;

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

@Override
public void start(Stage stage) {
    // Build UI
    VBox root = getUIs();
    scene = new Scene(root);
    stage.setScene(scene);

    // Add event handlers for the source and target
    this.addDnDEventHandlers();

    root.setStyle("-fx-padding: 10;" +
                  "-fx-border-style: solid inside;" +
                  "-fx-border-width: 2;" +
                  "-fx-border-insets: 5;" +
                  "-fx-border-radius: 5;" +
                  "-fx-border-color: blue;");
    stage.setTitle("Performing a Drag-and-Drop
Gesture");
    stage.show();
}

private VBox getUIs() {
    Label msgLbl = new Label(
        "Drag and drop an image, an image file, or an
image URL below.");

    // Set the size for the image view
    imageView.setFitWidth(300);
    imageView.setFitHeight(300);
    imageView.setSmooth(true);
    imageView.setPreserveRatio(true);

    clearBtn.setOnAction(e -> imageView.setImage(null));

    VBox box = new VBox(20, msgLbl, imageView,
    clearBtn);
    return box;
}

private void addDnDEventHandlers() {
    scene.setOnDragOver(this::dragOver);
    scene.setOnDragDropped(this::dragDropped);
}

private void dragOver(DragEvent e) {
    // You can drag an image, a URL or a file
    Dragboard dragboard = e.getDragboard();
}

```

```

        if (dragboard.hasImage() || dragboard.hasFiles() ||
dragboard.hasUrl()) {
            e.acceptTransferModes(TransferMode.ANY);
        }

        e.consume();
    }

private void dragDropped(DragEvent e) {
    boolean isCompleted = false;

    // Transfer the data to the target
    Dragboard dragboard = e.getDragboard();

    if (dragboard.hasImage()) {
        this.transferImage(dragboard.getImage());
        isCompleted = true;
    } else if (dragboard.hasFiles()) {
        isCompleted
= this.transferImageFile(dragboard.getFiles());
    } else if (dragboard.hasUrl()) {
        isCompleted
= this.transferImageUrl(dragboard.getUrl());
    } else {
        System.out.println("Dragboard does not contain
an image" +
                           " in the expected format:
Image, File, URL");
    }

    // Data transfer is not successful
    e.setDropCompleted(isCompleted);

    e.consume();
}

private void transferImage(Image image) {
    imageView.setImage(image);
}

private boolean transferImageFile(List<File> files) {
    // Look at the mime type of all file.
    // Use the first file having the mime type as
"image/xxx"
    for(File file : files) {
        String mimeType;
        try {
            mimeType
= Files.probeContentType(file.toPath());
            if (mimeType != null &&
mimeType.startsWith("image/")) {
                this.transferImageUrl(file.toURI()
.toURL().toExternalForm());
                return true;
            }
        }
    }
}

```

```
        }
        catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }

    return false;
}

private boolean transferImageUrl(String imageUrl) {
    try {
        imageView.setImage(new Image(imageUrl));
        return true;
    }
    catch (Exception e) {
        System.out.println(e.getMessage());
    }

    return false;
}
}
```

## Transferring Custom Data Types

You can transfer data in any format using the drag-and-drop gesture provided the data is `Serializable`. In this section, I will demonstrate how to transfer custom data. You will transfer an `ArrayList<Item>`. The `Item` class is shown in Listing 26-5; it is `Serializable`. The class is very simple. It contains one private field with its getter and setter methods.

### ***Listing 26-5.*** Using a Custom Data Type in Data Transfer

```
// Item.java
package com.jdojo.dnd;

import java.io.Serializable;

public class Item implements Serializable {
    private String name = "Unknown";

    public Item(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

```

@Override
public String toString() {
    return name;
}
}

```

The program in Listing 26-6 shows how to use a custom data format in a drag-and-drop gesture. It displays a window as shown in Figure 26-3. The window contains two `ListView`s. Initially, only one of the `ListView`s is populated with a list of items.

Both `ListView`s support multiple selection. You can select items in one `ListView` and drag and drop them into another `ListView`. The selected items will be copied or moved depending on the system-determined transfer mode. For example, on Windows, items will be moved by default. If you press the `Ctrl` key while dropping, the items will be copied instead.

### ***Listing 26-6.*** Transferring Custom Data Using a Drag-and-Drop Gesture

```

// CustomDataTransfer.java
package com.jdojo.dnd;

import java.util.ArrayList;
import java.util.List;
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.ListView;
import javafx.scene.control.SelectionMode;
import javafx.scene.input.ClipboardContent;
import javafx.scene.input.DataFormat;
import javafx.scene.input.DragEvent;
import javafx.scene.input.Dragboard;
import javafx.scene.input.MouseEvent;
import javafx.scene.input.TransferMode;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class CustomDataTransfer extends Application {
    ListView<Item> lv1 = new ListView<>();
    ListView<Item> lv2 = new ListView<>();

    // Our custom Data Format
    static final DataFormat ITEM_LIST = new
    DataFormat("jdojo/itemlist");

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

```

```

@Override
public void start(Stage stage) {
    // Build the UI
    GridPane root = getUIs();

    // Add event handlers for the source and target
    // text fields of the the DnD operation
    this.addDnDEventHandlers();

    root.setStyle("-fx-padding: 10;" +
                  "-fx-border-style: solid inside;" +
                  "-fx-border-width: 2;" +
                  "-fx-border-insets: 5;" +
                  "-fx-border-radius: 5;" +
                  "-fx-border-color: blue;");

    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("Drag-and-Drop Test");
    stage.show();
}

private GridPane getUIs() {
    Label msgLbl = new Label("Select one or more items
from a list, " +
                           "drag and drop them to another
list");

    lv1.setPrefSize(200, 200);
    lv2.setPrefSize(200, 200);
    lv1.getItems().addAll(this.getList());

    // Allow multi-select in lists
    lv1.getSelectionModel().setSelectionMode(SelectionMode
de.MULTIPLE);
    lv2.getSelectionModel().setSelectionMode(SelectionMode
de.MULTIPLE);

    GridPane pane = new GridPane();
    pane.setHgap(10);
    pane.setVgap(10);
    pane.add(msgLbl, 0, 0, 3, 1);
    pane.addRow(1, new Label("List 1:"), new
Label("List 2:" ));
    pane.addRow(2, lv1, lv2);
    return pane;
}

private ObservableList<Item> getList() {
    ObservableList<Item> list
= FXCollections.<Item>observableArrayList();
    list.addAll(new Item("Apple"), new Item("Orange"),
               new Item("Papaya"), new Item("Mango"),
               new Item("Grape"), new Item("Guava"));
    return list;
}

```

```

    }

private void addDnDEventHandlers() {
    lv1.setOnDragDetected(e -> dragDetected(e, lv1));
    lv2.setOnDragDetected(e -> dragDetected(e, lv2));

    lv1.setOnDragOver(e -> dragOver(e, lv1));
    lv2.setOnDragOver(e -> dragOver(e, lv2));

    lv1.setOnDragDropped(e -> dragDropped(e, lv1));
    lv2.setOnDragDropped(e -> dragDropped(e, lv2));

    lv1.setOnDragDone(e -> dragDone(e, lv1));
    lv2.setOnDragDone(e -> dragDone(e, lv2));
}

private void dragDetected(MouseEvent e, ListView<Item> listView) {
    // Make sure at least one item is selected
    int selectedCount
= listView.getSelectionModel().getSelectedIndices().size();
    if (selectedCount == 0) {
        e.consume();
        return;
    }

    // Initiate a drag-and-drop gesture
    Dragboard dragboard
= listView.startDragAndDrop(TransferMode.COPY_OR_MOVE);

    // Put the the selected items to the dragboard
    ArrayList<Item> selectedItems
= this.getSelectedItems(listView);
    ClipboardContent content = new ClipboardContent();
    content.put(ITEM_LIST, selectedItems);
    dragboard.setContent(content);

    e.consume();
}

private void dragOver(DragEvent e, ListView<Item> listView)
{
    // If drag board has an ITEM_LIST and it is not
being dragged
    // over itself, we accept the MOVE transfer mode
    Dragboard dragboard = e.getDragboard();

    if (e.getGestureSource() != listView &&
        dragboard.hasContent(ITEM_LIST)) {
        e.acceptTransferModes(TransferMode.COPY_OR_MOV
E);
    }

    e.consume();
}

```

```

@SuppressWarnings("unchecked")
private void dragDropped(DragEvent e, ListView<Item>
listView) {
    boolean dragCompleted = false;

    // Transfer the data to the target
    Dragboard dragboard = e.getDragboard();

    if(dragboard.hasContent(ITEM_LIST)) {
        ArrayList<Item> list
= (ArrayList<Item>)dragboard.getContent(ITEM_LIST);
        listView.getItems().addAll(list);

        // Data transfer is successful
        dragCompleted = true;
    }

    // Data transfer is not successful
    e.setDropCompleted(dragCompleted);

    e.consume();
}

private void dragDone(DragEvent e, ListView<Item> listView)
{
    // Check how data was transferred to the target
    // If it was moved, clear the selected items
    TransferMode tm = e.getTransferMode();

    if (tm == TransferMode.MOVE) {
        removeSelectedItems(listView);
    }

    e.consume();
}

private ArrayList<Item> getSelectedItems(ListView<Item>
listView) {
    // Return the list of selected item in an
ArrayList, so it is
    // serializable and can be stored in a Dragboard.
    ArrayList<Item> list =
        new
ArrayList<>(listView.getSelectionModel().getSelectedItems());
    return list;
}

private void removeSelectedItems(ListView<Item> listView) {
    // Get all selected items in a separate list to
    // avoid the shared list issue
    List<Item> selectedList = new ArrayList<>();
    for(Item item
: listView.getSelectionModel().getSelectedItems()) {
        selectedList.add(item);
    }
}

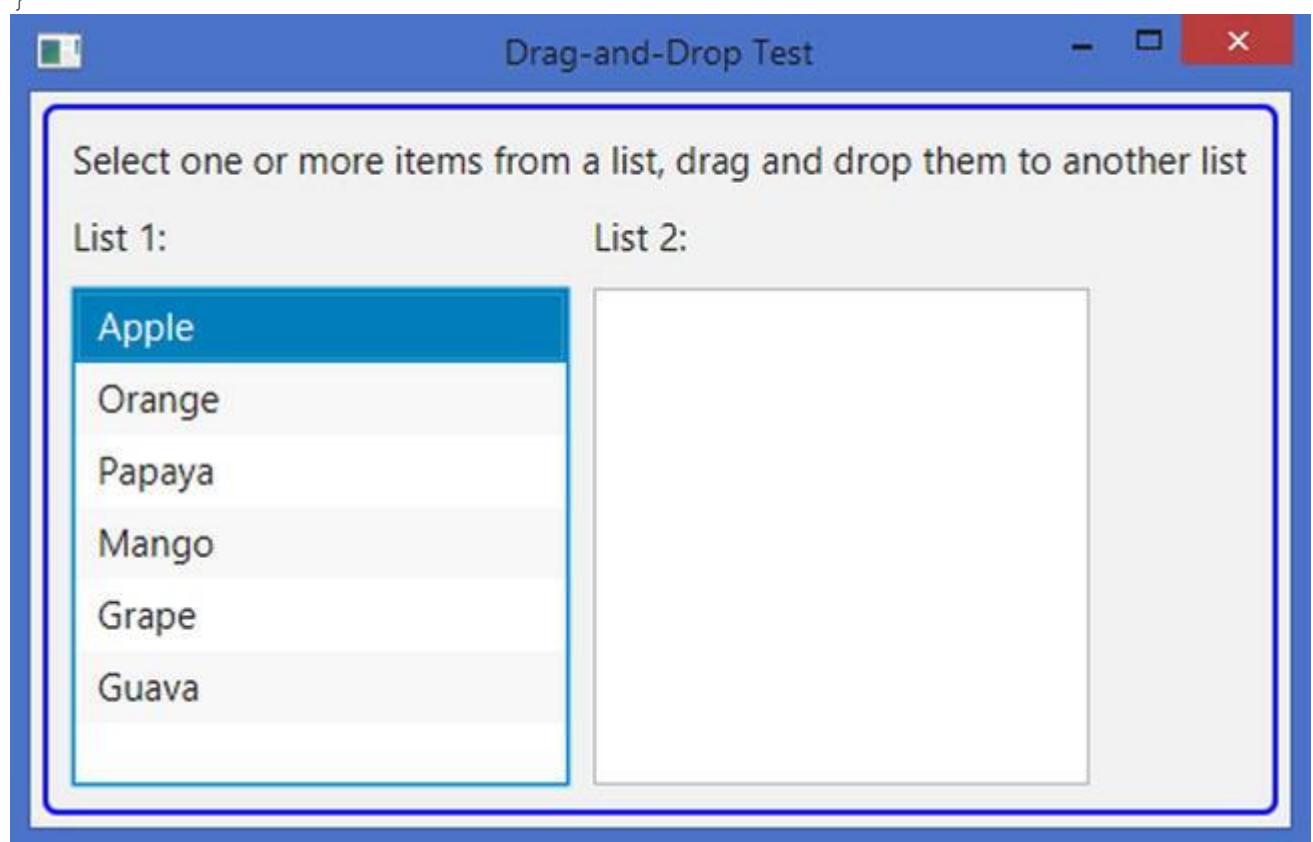
```

```

        // Clear the selection
        listView.getSelectionModel().clearSelection();

        // Remove items from the selected list
        listView.getItems().removeAll(selectedList);
    }
}

```



**Figure 26-3.** Transferring a list of selected items between two `ListViews`

Most of the program is similar to what you have seen before. The difference is in how you store and retrieve the `ArrayList<Item>` in the dragboard.

You define a new data format for this data transfer because the data do not fit into any of the categories available as the constants in the `DataFormat` class. You have to define the data as constants, as in the following code:

```

// Our custom Data Format
static final DataFormat ITEM_LIST = new
DataFormat("jdojo/itemlist");

```

Now you have given a unique mime type `jdojo/itemlist` for the data format.

In the drag-detected event, you need to store the list of selected items onto the dragboard. The following snippet of code in

the `dragDetected()` method stores the job. Notice that you have used the new data format while storing the data on the dragboard.

```
ArrayList<Item> selectedItems = this.getSelectedItems(listView);
ClipboardContent content = new ClipboardContent();
content.put(ITEM_LIST, selectedItems);
dragboard.setContent(content);
```

In the drag-over event, if the `ListView` is not being dragged over itself and the dragboard contains data in the `ITEM_LIST` data format, the `ListView` declares that it accepts a `COPY` or `MOVE` transfer. The following snippet of code in the `dragOver()` method does the job:

```
Dragboard dragboard = e.getDragboard();
if (e.getGestureSource() != listView &&
dragboard.hasContent(ITEM_LIST)) {
    e.acceptTransferModes(TransferMode.COPY_OR_MOVE);
}
```

Finally, you need to read the data from the dragboard when the source is dropped on the target. You need to use the `getContent()` method of the dragboard specifying the `ITEM_LIST` as the data format. The returned result needs to be cast to the `ArrayList<Item>`. The following snippet of code in the `dragDropped()` method does the job:

```
Dragboard dragboard = e.getDragboard();
if (dragboard.hasContent(ITEM_LIST)) {
    ArrayList<Item> list
= (ArrayList<Item>) dragboard.getContent(ITEM_LIST);
    listView.getItems().addAll(list);

    // Data transfer is successful
    dragCompleted = true;
}
```

Finally, in the drag-done event handler, which is implemented in the `dragDone()` method, you remove the selected items from the source `ListView` if `MOVE` was used as the transfer mode. Notice that you have used an `ArrayList<Item>`, as both the `ArrayList` and `Item` classes are serializable.

## Summary

A press-drag-release gesture is a user action of pressing a mouse button, dragging the mouse with the pressed button, and releasing the button. The gesture can be initiated on a scene or a node. Several nodes and scenes may participate in a single press-drag-release gesture. The gesture is capable of generating different types of events and delivering those events to different nodes. The type of generated events and the nodes receiving the events depend on the purpose of the gesture.

JavaFX supports three types of drag gestures: a simple press-drag-release gesture, a full press-drag-release gesture, and a drag-and-drop gesture.

The simple press-drag-release gesture is the default drag gesture. It is used when the drag gesture involves only one node—the node on which the gesture was initiated. During the drag gesture, all `MouseEvent` types—mouse-drag entered, mouse-drag over, mouse-drag exited, mouse, and mouse-drag released—are delivered only to the gesture source node.

When the source node of a drag gesture receives the drag-detected event, you can start a full press-drag-release gesture by calling the `startFullDrag()` method on the source node.

The `startFullDrag()` method exists in both `Node` and `Scene` classes, allowing you to start a full press-drag-release gesture for a node and a scene.

The third type of drag gesture is called a drag-and-drop gesture, which is a user action combining the mouse movement with a pressed mouse button. It is used to transfer data from the gesture source to a gesture target. In a drag-and-drop gesture, the data can be transferred in three modes: Copy, Move, and Link. The copy mode indicates that the data will be copied from the gesture source to the gesture target. The move mode indicates that the data will be moved from the gesture source to the gesture target. The link mode indicates that the gesture target will create a link (or reference) to the data being transferred. The actual meaning of “link” depends on the application.

In a drag-and-drop data transfer, the gesture source and the gesture target do not know each other—they may even belong to two different applications. A dragboard acts as an intermediary between the gesture source and the gesture target. A dragboard is the storage device to hold the data being transferred. The gesture source places the data onto a dragboard; the dragboard is made available to the gesture target, so it can inspect the type of content that is available for the transfer. When the gesture target is ready to transfer the data, it gets the data from the dragboard.

Using a drag-and-drop gesture, the data transfer takes place in three steps: initiating the drag-and-drop gesture by the source, detecting the drag gesture by the target, and dropping the source onto the target. Different types of events are generated for the source and target nodes during this gesture. You can also provide visual clues by showing icons during the drag-and-drop gesture. The drag-and-drop gesture supports transferring of any type of data, provided the data are serializable.

The next chapter discusses how to handle concurrent operations in JavaFX.

## CHAPTER 27



### Understanding Concurrency in JavaFX

In this chapter, you will learn:

- Why you need a concurrency framework in JavaFX
- How the `Worker<V>` interface represents a concurrent task
- How to run a one-time task
- How to run a reusable task
- How to run a scheduled task

### The Need for a Concurrency Framework

Java (including JavaFX) GUI (graphical user interface) applications are inherently multithreaded. Multiple threads perform different tasks to keep the UI in sync with the user actions. JavaFX, like Swing and AWT, uses a single thread, called JavaFX Application Thread, to process all UI events. The nodes representing UI in a scene graph are not thread-safe. Designing nodes that are not thread-safe has advantages and disadvantages. They are faster, as no synchronization is involved. The disadvantage is that they need to be accessed from a single thread to avoid being in an illegal state. JavaFX puts a restriction that a live scene graph must be accessed from one and only one thread, the JavaFX Application Thread. This restriction indirectly imposes another restriction that a UI event should not process a long-running task, as it will make the application unresponsive. The user will get the impression that the application is hung.

The program in Listing 27-1 displays a window as shown in Figure 27-1. It contains three controls.

- A `Label` to display the progress of a task
- A `Start` button to start the task
- An `Exit` button to exit the application

#### ***Listing 27-1.*** Performing a Long-Running Task in an Event Handler

```
// UnresponsiveUI.java
package com.jdojo.concurrent;

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

```
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

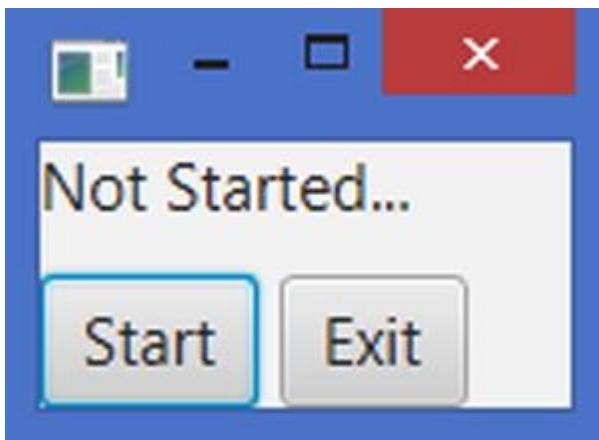
public class UnresponsiveUI extends Application {
    Label statusLbl = new Label("Not Started...");
    Button startBtn = new Button("Start");
    Button exitBtn = new Button("Exit");

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

    @Override
    public void start(Stage stage) {
        // Add event handlers to the buttons
        startBtn.setOnAction(e -> runTask());
        exitBtn.setOnAction(e -> stage.close());

        HBox buttonBox = new HBox(5, startBtn, exitBtn);
        VBox root = new VBox(10, statusLbl, buttonBox);
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("An Unresponsive UI");
        stage.show();
    }

    public void runTask() {
        for(int i = 1; i <= 10; i++) {
            try {
                String status = "Processing " + i + " of
" + 10;
                statusLbl.setText(status);
                System.out.println(status);
                Thread.sleep(1000);
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```



**Figure 27-1.** An example of an unresponsive UI

The program is very simple. When you click the *Start* button, a task lasting for 10 seconds is started. The logic for the task is in the `runTask()` method, which simply runs a loop ten times. Inside the loop, the task lets the current thread, which is the JavaFX Application Thread, sleep for 1 second. The program has two problems.

Click the *Start* button and immediately try to click the *Exit* button. Clicking the *Exit* button has no effect until the task finishes. Once you click the *Start* button, you cannot do anything else on the window, except to wait for 10 seconds for the task to finish. That is, the application becomes unresponsive for 10 seconds. This is the reason you named the class `UnresponsiveUI`.

Inside the loop in the `runTask()` method, the program prints the status of the task on the standard output and displays the same in the `Label` in the window. You see the status updated on the standard output, but not in the `Label`.

It is repeated to emphasize that all UI event handlers in JavaFX run on a single thread, which is the JavaFX Application Thread. When the *Start* button is clicked, the `runTask()` method is executed in the JavaFX Application Thread. When the *Exit* button is clicked while the task is running, an `ActionEvent` event for the *Exit* button is generated and queued on the JavaFX Application Thread.

The `ActionEvent` handler for the *Exit* button is run on the same thread after the thread is done running the `runTask()` method as part of the `ActionEvent` handler for the *Start* button.

A pulse event is generated when the scene graph is updated. The pulse event handler is also run on the JavaFX Application Thread. Inside the loop, the `text` property of the `Label` was updated ten times, which generated the pulse events. However, the scene graph was not refreshed

to show the latest text for the `Label`, as the JavaFX Application Thread was busy running the task and it did not run the pulse event handlers.

Both problems arise because there is only one thread to process all UI event handlers and you ran a long-running task in the `ActionEvent` handler for the `Start` button.

What is the solution? You have only one option. You cannot change the single-threaded model for handling the UI events. You must not run long-running tasks in the event handlers. Sometimes, it is a business need to process a big job as part of a user action. The solution is to run the long-running tasks in one or more background threads, instead of in the JavaFX Application Thread.

The program in Listing 27-2 is your first, incorrect attempt to provide a solution. The `ActionEventhandler` for the `Start` button calls the `startTask()` method, which creates a new thread and runs the `runTask()` method in the new thread.

### ***Listing 27-2.*** A Program Accessing a Live Scene Graph from a Non-JavaFX Application Thread

```
// BadUI.java
package com.jdojo.concurrent;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class BadUI extends Application {
    Label statusLbl = new Label("Not Started...");
    Button startBtn = new Button("Start");
    Button exitBtn = new Button("Exit");

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

    @Override
    public void start(Stage stage) {
        // Add event handlers to the buttons
        startBtn.setOnAction(e -> startTask());
        exitBtn.setOnAction(e -> stage.close());

        HBox buttonBox = new HBox(5, startBtn, exitBtn);
        VBox root = new VBox(10, statusLbl, buttonBox);
        Scene scene = new Scene(root);
        stage.setScene(scene);
    }

    private void startTask() {
        // This code runs in a separate thread
        // and updates the status label
        Platform.runLater(() -> statusLbl.setText("Started"));
    }
}
```

```
        stage.setTitle("A Bad UI");
        stage.show();
    }

    public void startTask() {
        // Create a Runnable
        Runnable task = () -> runTask();

        // Run the task in a background thread
        Thread backgroundThread = new Thread(task);

        // Terminate the running thread if the application
        // exits
        backgroundThread.setDaemon(true);

        // Start the thread
        backgroundThread.start();
    }

    public void runTask() {
        for(int i = 1; i <= 10; i++) {
            try {
                String status = "Processing " + i + " of
" + 10;
                statusLbl.setText(status);
                System.out.println(status);
                Thread.sleep(1000);
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Run the program and click the *Start* button. A runtime exception is thrown. The partial stack trace of the exception is as follows:

```
Exception in thread "Thread-4" java.lang.IllegalStateException:  
Not on FX application thread; currentThread = Thread-4  
    at  
com.sun.javafx.tk.Toolkit.checkFxUserThread(Toolkit.java:209)  
    at  
com.sun.javafx.tk.quantum.QuantumToolkit.checkFxUserThread(QuantumToolkit.java:393)...  
at com.jdojo.concurrent.BadUI.runTask(BadUI.java:47) ...
```

The following statement in the `runTask()` method generated the exception

```
statusLbl.setText(status);
```

The JavaFX runtime checks that a live scene must be accessed from the JavaFX Application Thread. The `runTask()` method is run on a new thread, named Thread-4 as shown in the stack trace, which is not the JavaFX Application Thread. The foregoing statement sets

the `text` property for the `Label`, which is part of a live scene graph, from the thread other than the JavaFX Application Thread, which is not permissible.

How do you access a live scene graph from a thread other than the JavaFX Application Thread? The simple answer is that you cannot. The complex answer is that when a thread wants to access a live scene graph, it needs to run the part of the code that accesses the scene graph in the JavaFX Application Thread. The `Platform` class in the `javafx.application` package provides two static methods to work with the JavaFX application Thread.

- `public static boolean isFxApplicationThread()`
- `public static void runLater(Runnable runnable)`

The `isFxApplicationThread()` method returns true if the thread calling this method is the JavaFX Application Thread. Otherwise, it returns false.

The `runLater()` method schedules the specified `Runnable` to be run on the JavaFX Application Thread at some unspecified time in future.

**Tip** If you have experience working with Swing, the `Platform.runLater()` in JavaFX is the counterpart of the `SwingUtilities.invokeLater()` in Swing.

Let us fix the problem in the `BadUI` application. The program in Listing 27-3 is the correct implementation of the logic to access the live scene graph. Figure 27-2 shows a snapshot of the window displayed by the program.

### ***Listing 27-3.*** A Responsive UI That Runs Long-Running Tasks in a Background Thread

```
// ResponsiveUI.java
package com.jdojo.concurrent;

import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ResponsiveUI extends Application {
```

```

Label statusLbl = new Label("Not Started...");
Button startBtn = new Button("Start");
Button exitBtn = new Button("Exit");

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

@Override
public void start(Stage stage) {
    // Add event handlers to the buttons
    startBtn.setOnAction(e -> startTask());
    exitBtn.setOnAction(e -> stage.close());

    HBox buttonBox = new HBox(5, startBtn, exitBtn);
    VBox root = new VBox(10, statusLbl, buttonBox);
    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("A Responsive UI");
    stage.show();
}

public void startTask() {
    // Create a Runnable
    Runnable task = () -> runTask();

    // Run the task in a background thread
    Thread backgroundThread = new Thread(task);

    // Terminate the running thread if the application
    exits
    backgroundThread.setDaemon(true);

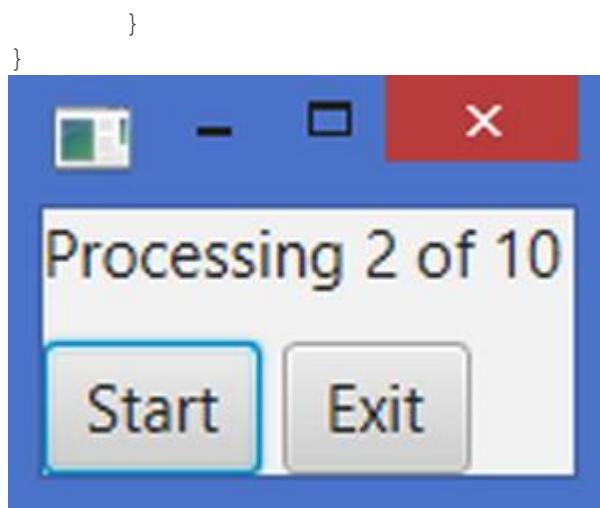
    // Start the thread
    backgroundThread.start();
}

public void runTask() {
    for(int i = 1; i <= 10; i++) {
        try {
            String status = "Processing " + i + " of
" + 10;

            // Update the Label on the JavaFx
            Application Thread
                Platform.runLater(() ->
statusLbl.setText(status));

            System.out.println(status);
            Thread.sleep(1000);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```



**Figure 27-2.** A UI that runs a task in a background thread and updates the live scene graph correctly

The program replaces the statement

```
statusLbl.setText(status);
```

in the `BadUI` class with the statement

```
// Update the Label on the JavaFX Application Thread
Platform.runLater(() -> statusLbl.setText(status));
```

Now, setting the `text` property for the `Label` takes place on the JavaFX Application Thread. The `ActionEvent` handler of the *Start* button runs the task in a background thread, thus freeing up the JavaFX Application Thread to handle user actions. The status of the task is updated in the `Label` regularly. You can click the *Exit* button while the task is being processed.

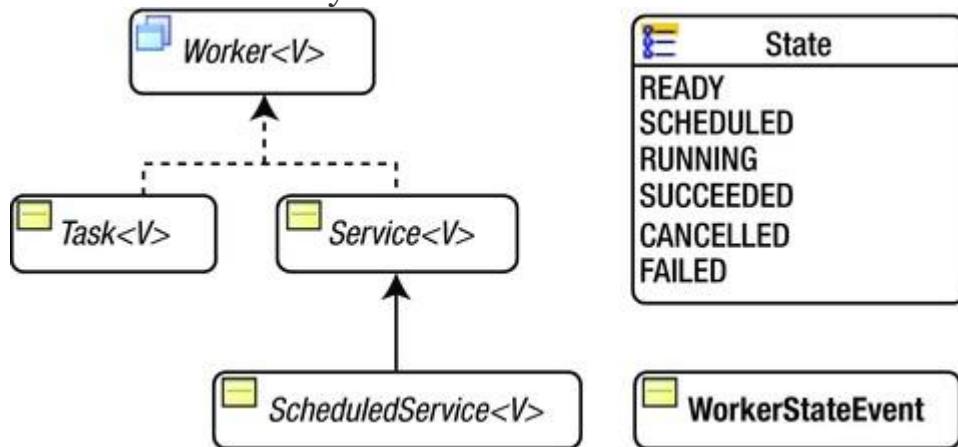
Did you overcome the restrictions imposed by the event-dispatching threading model of the JavaFX? The answer is yes and no. You used a trivial example to demonstrate the problem. You have solved the trivial problem. However, in a real world, performing a long-running task in a GUI application is not so trivial. For example, your task-running logic and the UI are tightly coupled as you are referencing the `Label` inside the `runTask()` method, which is not desirable in a real world. Your task does not return a result, nor does it have a reliable mechanism to handle errors that may occur. Your task cannot be reliably cancelled, restarted, or scheduled to be run at a future time.

The JavaFX concurrency framework has answers to all these questions. The framework provides a reliable way of running a task in one or multiple background threads and publishing the status and the result of the task in a GUI application. The framework is the topic of discussion in this chapter. I have taken several pages just to make the case for a concurrency framework in JavaFX. If you understand the

background of the problem as presented in this section, understanding the framework will be easy.

## Understanding the Concurrent Framework API

Java 5 added a comprehensive concurrency framework to the Java programming language through the libraries in the `java.util.concurrent` package. The JavaFX concurrency framework is very small. It is built on top of the Java language concurrency framework keeping in mind that it will be used in a GUI environment. Figure 27-3 shows a class diagram of the classes in the JavaFX concurrency framework.



**Figure 27-3.** A class diagram for classes in the JavaFX Concurrency Framework

The framework consists of one interface, four classes, and one enum.

An instance of the `Worker` interface represents a task that needs to be performed in one or more background threads. The state of the task is observable from the JavaFX Application Thread.

The `Task`, `Service`, and `ScheduledService` classes implement the `Worker` interface. They represent different types of tasks. They are abstract classes. An instance of the `Task` class represents a one-shot task. A `Task` cannot be reused. An instance of the `Service` class represents a reusable task. The `ScheduledService` class inherits from the `Service` class. A `ScheduledService` is a task that can be scheduled to run repeatedly after a specified interval.

The constants in the `Worker.State` enum represent different states of a `Worker`.

An instance of the `WorkerStateEvent` class represents an event that occurs as the state of a `Worker` changes. You can add event handlers to all three types of tasks to listen to the change in their states.

## Understanding the `Worker<V>` Interface

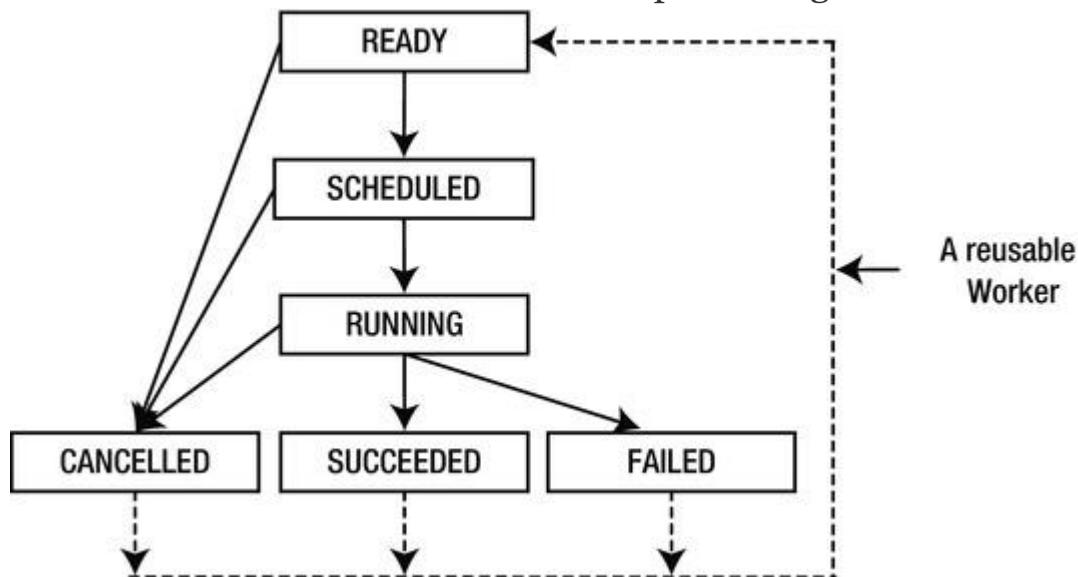
The `Worker<V>` interface provides the specification for any task performed by the JavaFX concurrency framework. A `Worker` is a task that is performed in one or more background threads. The generic parameter `V` is the data type of the result of the `Worker`. Use `Void` as the generic parameter if the `Worker` does not produce a result. The state of the task is observable. The state of the task is published on the JavaFX Application Thread, making it possible for the task to communicate with the scene graph, as is commonly required in a GUI application.

### State Transitions for a Worker

During the life cycle, a `Worker` transitions through different states. The constants in the `Worker.State` enum represent the valid states of a `Worker`.

- `Worker.State.READY`
- `Worker.State.SCHEDULED`
- `Worker.State.RUNNING`
- `Worker.State.SUCCEEDED`
- `Worker.State.CANCELLED`
- `Worker.State.FAILED`

Figure 27-4 shows the possible state transitions of a `Worker` with the `Worker.State` enum constants representing the states.



**Figure 27-4.** Possible state transition paths for a `Worker`

When a `Worker` is created, it is in the `READY` state. It transitions to the `SCHEDULED` state, before it starts executing. When it starts running, it is in the `RUNNING` state. Upon successful completion,

a Worker transitions from the RUNNING state to the SUCCEEDED state. If the Worker throws an exception during its execution, it transitions to the FAILED state. A Worker may be cancelled using the cancel() method. It may transition to the CANCELLED state from the READY, SCHEDULED, and RUNNING states. These are the normal state transitions for a one-shot Worker.

A reusable Worker may transition from the CANCELLED, SUCCEEDED, and FAILED states to the READY state as shown in the figure by dashed lines.

### Properties of a Worker

The Worker interface contains nine read-only properties that represent the internal state of the task.

- title
- message
- running
- state
- progress
- workDone
- totalWork
- value
- exception

When you create a Worker, you will have a chance to specify these properties. The properties can also be updated as the task progresses.

The title property represents the title for the task. Suppose a task generates prime numbers. You may give the task a title “Prime Number Generator.”

The message property represents a detailed message during the task processing. Suppose a task generates several prime numbers; you may want to give feedback to the user at a regular interval or at appropriate times with a message such as “Generating X of Y prime numbers.”

The running property tells whether the Worker is running. It is true when the Worker is in the SCHEDULED or RUNNING states. Otherwise, it is false.

The state property specifies the state of the Worker. Its value is one of the constants of the Worker.State enum.

The totalWork, workDone, and progress properties represent the progress of the task. The totalWork is the total amount of work to be done. The workDone is the amount of work that has been done.

The progress is the ratio of `workDone` and `totalWork`. They are set to -1.0 if their values are not known.

The `value` property represents the result of the task. Its value is non-null only when the `Worker` finishes successfully reaching the `SUCCEEDED` state. Sometimes, a task may not produce a result. In those cases, the generic parameter `V` would be `Void` and the `value` property will always be `null`.

A task may fail by throwing an exception. The `exception` property represents the exception that is thrown during the processing of the task. It is non-null only when the state of the `Worker` is `FAILED`. It is of the type `Throwable`.

Typically, when a task is in progress, you want to display the task details in a scene graph. The concurrency framework makes sure that the properties of a `Worker` are updated on the JavaFX Application Thread. Therefore, it is fine to bind the properties of the UI elements in a scene graph to these properties. You can also add `Invalidation` and `ChangeListener` to these properties and access a live scene graph from inside those listeners.

In subsequent sections, you will discuss specific implementations of the `Worker` interface. Let us create a reusable GUI to use in all examples. The GUI is based on a `Worker` to display the current values of its properties.

## Utility Classes for Examples

Let us create the reusable GUI and non-GUI parts of the programs to use in examples in the subsequent sections. The `WorkerStateUI` class in Listing 27-4 builds a `GridPane` to display all properties of a `Worker`. It is used with a `Worker<ObservableList<Long>>`. It displays the properties of a `Worker` by UI elements to them. You can bind properties of a `Worker` to the UI elements by passing a `Worker` to the constructor or calling the `bindToWorker()` method.

### **Listing 27-4.** A Utility Class to Build UI Displaying the Properties of a Worker

```
// WorkerStateUI.java
package com.jdojo.concurrent;

import javafx.beans.binding.When;
import javafx.collections.ObservableList;
import javafx.concurrent.Worker;
import javafx.scene.control.Label;
import javafx.scene.control.ProgressBar;
```

```

import javafx.scene.control.TextArea;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;

public class WorkerStateUI extends GridPane {
    private final Label title = new Label("");
    private final Label message = new Label("");
    private final Label running = new Label("");
    private final Label state = new Label("");
    private final Label totalWork = new Label("");
    private final Label workDone = new Label("");
    private final Label progress = new Label("");
    private final TextArea value = new TextArea("");
    private final TextArea exception = new TextArea("");
    private final ProgressBar progressBar = new ProgressBar();

    public WorkerStateUI() {
        addUI();
    }

    public WorkerStateUI(Worker<ObservableList<Long>> worker) {
        addUI();
        bindToWorker(worker);
    }

    private void addUI() {
        value.setPrefColumnCount(20);
        value.setPrefRowCount(3);
        exception.setPrefColumnCount(20);
        exception.setPrefRowCount(3);
        this.setHgap(5);
        this.setVgap(5);
        addRow(0, new Label("Title:"), title);
        addRow(1, new Label("Message:"), message);
        addRow(2, new Label("Running:"), running);
        addRow(3, new Label("State:"), state);
        addRow(4, new Label("Total Work:"), totalWork);
        addRow(5, new Label("Work Done:"), workDone);
        addRow(6, new Label("Progress:"), new HBox(2,
progressBar, progress));
        addRow(7, new Label("Value:"), value);
        addRow(8, new Label("Exception:"), exception);
    }

    public void bindToWorker(final Worker<ObservableList<Long>>
worker) {
        // Bind Labels to the properties of the worker
        title.textProperty().bind(worker.titleProperty());
        message.textProperty().bind(worker.messageProperty());
    }
    running.textProperty().bind(worker.runningProperty()
.asString());
    state.textProperty().bind(worker.stateProperty().asS
tring());
}

```

```

        totalWork.textProperty().bind(new
When(worker.totalWorkProperty().isEqualTo(-1))
        .then("Unknown")
        .otherwise(worker.totalWorkProperty().asString()));
        workDone.textProperty().bind(new
When(worker.workDoneProperty().isEqualTo(-1))
        .then("Unknown")
        .otherwise(worker.workDoneProperty().asString()));
        progress.textProperty().bind(new
When(worker.progressProperty().isEqualTo(-1))
        .then("Unknown")
        .otherwise(worker.progressProperty().multipl
y(100.0)
                .asString("%.2f%%")));
        progressBar.progressProperty().bind(worker.progressP
roperty());
        value.textProperty().bind(worker.valueProperty().ass
tring()));

        // Display the exception message when an exception
occurs in the worker
        worker.exceptionProperty().addListener((prop,
oldValue, newValue) -> {
            if (newValue != null) {
                exception.setText(newValue.getMessage())
;
            } else {
                exception.setText("");
            }
        });
    }
}

```

The PrimeUtil class in Listing 27-5 is a utility class to check whether a number is a prime number.

### ***Listing 27-5.*** A Utility Class to Work with Prime Numbers

```

// PrimeUtil.java
package com.jdojo.concurrent;

public class PrimeUtil {
    public static boolean isPrime(long num) {
        if (num <= 1 || num % 2 == 0) {
            return false;
        }

        int upperDivisor = (int) Math.ceil(Math.sqrt(num));
        for (int divisor = 3; divisor <= upperDivisor;
divisor += 2) {
            if (num % divisor == 0) {
                return false;
            }
        }
    }
}

```

```

        }
    }
    return true;
}
}

```

## Using the Task<V> Class

An instance of the `Task<V>` class represents a one-time task. Once the task is completed, cancelled, or failed, it cannot be restarted.

The `Task<V>` class implements the `Worker<V>` interface. Therefore, all properties and methods specified by the `Worker<V>` interface are available in the `Task<V>` class.

The `Task<V>` class inherits from the `FutureTask<V>` class, which is part of the Java concurrency framework.

The `FutureTask<V>` implements

`the Future<V>, RunnableFuture<V>, and Runnable interfaces.`

Therefore, a `Task<V>` also implements all these interfaces.

### Creating a Task

How do you create a `Task<V>`? Creating a `Task<V>` is easy. You need to subclass the `Task<V>` class and provide an implementation for the abstract method `call()`. The `call()` method contains the logic to perform the task. The following snippet of code shows the skeleton of a `Task` implementation:

```

// A Task that produces an ObservableList<Long>
public class PrimeFinderTask extends Task<ObservableList<Long>> {
    @Override
    protected ObservableList<Long>> call() {
        // Implement the task logic here...
    }
}

```

### Updating Task Properties

Typically, you would want to update the properties of the task as it progresses. The properties must be updated and read on the JavaFX Application Thread, so they can be observed safely in a GUI environment. The `Task<V>` class provides special methods to update some of its properties.

- `protected void updateMessage(String message)`
- `protected void updateProgress(double workDone, double totalWork)`

- `protected void updateProgress(long workDone, long totalWork)`
- `protected void updateTitle(String title)`
- `protected void updateValue(V value)`

You provide the values for the `workDone` and the `totalWork` properties to the `updateProgress()` method. The `progress` property will be set to `workDone/totalWork`. The method throws a runtime exception if the `workDone` is greater than the `totalWork` or both are less than -1.0.

Sometimes, you may want to publish partial results of a task in its `value` property. The `updateValue()` method is used for this purpose. The final result of a task is the return value of its `call()` method.

All `updateXXX()` methods are executed on the JavaFX application Thread. Their names indicate the property they update. They are safe to be called from the `call()` method of the Task. If you want to update the properties of the Task from the `call()` method directly, you need to wrap the code inside a `Platform.runLater()` call.

## Listening to Task Transition Events

The `Task` class contains the following properties to let you set event handlers for its state transitions:

- `onCancelled`
- `onFailed`
- `onRunning`
- `onScheduled`
- `onSucceeded`

The following snippet of code adds an `onSucceeded` event handler, which would be called when the task transitions to the `SUCCEEDED` state:

```
Task<ObservableList<Long>> task = create a task...
task.setOnSucceeded(e -> {
    System.out.println("The task finished. Let us party!")
});
```

## Cancelling a Task

Use one of the following two `cancel()` methods to cancel a task:

- `public final boolean cancel()`

- `public boolean cancel(boolean  
mayInterruptIfRunning)`

The first version removes the task from the execution queue or stops its execution. The second version lets you specify whether the thread running the task be interrupted. Make sure to handle the `InterruptedException` inside the `call()` method. Once you detect this exception, you need to finish the `call()` method quickly. Otherwise, the call to `cancel(true)` may not cancel the task reliably. The `cancel()` method may be called from any thread.

The following methods of the `Task` are called when it reaches a specific state:

- `protected void scheduled()`
- `protected void running()`
- `protected void succeeded()`
- `protected void cancelled()`
- `protected void failed()`

Their implementations in the `Task` class are empty. They are meant to be overridden by the subclasses.

## Running a Task

A `Task` is `Runnable` as well as a `FutureTask`. To run it, you can use a **background thread or an ExecutorService**.

```
// Schedule the task on a background thread
Thread backgroundThread = new Thread(task);
backgroundThread.setDaemon(true);
backgroundThread.start();

// Use the executor service to schedule the task
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.submit(task);
```

## A Prime Finder Task Example

It is time to see a `Task` in action. The program in Listing 27-6 is an implementation of the `Task<ObservableList<Long>>`. It checks for prime numbers between the specified `lowerLimit` and `upperLimit`. It returns all the numbers in the range. Notice that the task thread sleeps for a short time before checking a number for a prime number. This is done to give the user an impression of a long-running task. It is not needed in a real world application. The `call()` method handles

an `InterruptedException` and finishes the task if the task was interrupted as part of a cancellation request.

The call to the method `updateValue()` needs little explanation.

```
updateValue(FXCollections.<Long>unmodifiableObservableList(result
s));
```

Every time a prime number is found, the results list is updated. The foregoing statement wraps the results list in an unmodifiable observable list and publishes it for the client. This gives the client access to the partial results of the task. This is a quick and dirty way of publishing the partial results. If the `call()` method returns a primitive value, it is fine to call the `updateValue()` method repeatedly.

**Tip** In this case, you are creating a new unmodifiable list every time you find a new prime number, which is not acceptable in a production environment for performance reasons. The efficient way of publishing the partial results would be to declare a read-only property for the Task; update the read-only property regularly on JavaFX Application Thread; let the client bind to the read-only property to see the partial results.

### ***Listing 27-6.*** Finding Prime Numbers Using a Task<Long>

```
// PrimeFinderTask.java
package com.jdojo.concurrent;

import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.concurrent.Task;

public class PrimeFinderTask extends Task<ObservableList<Long>> {
    private long lowerLimit = 1;
    private long upperLimit = 30;
    private long sleepTimeInMillis = 500;

    public PrimeFinderTask() {
    }

    public PrimeFinderTask(long lowerLimit, long upperLimit) {
        this.lowerLimit = lowerLimit;
        this.upperLimit = upperLimit;
    }

    public PrimeFinderTask(long lowerLimit,
                          long upperLimit,
                          long sleepTimeInMillis) {
        this(lowerLimit, upperLimit);
        this.sleepTimeInMillis = sleepTimeInMillis;
    }

    // The task implementation
    @Override
    protected ObservableList<Long> call() {
```

```

        // An observable list to represent the results
        final ObservableList<Long> results =
            FXCollections.<Long>observableArrayList(
);

        // Update the title
        this.updateTitle("Prime Number Finder Task");

        long count = this.upperLimit - this.lowerLimit + 1;
        long counter = 0;

        // Find the prime numbers
        for (long i = lowerLimit; i <= upperLimit; i++) {
            // Check if the task is cancelled
            if (this.isCancelled()) {
                break;
            }

            // Increment the counter
            counter++;

            // Update message
            this.updateMessage("Checking " + i + " for
a prime number");

            // Sleep for some time
            try {
                Thread.sleep(this.sleepTimeInMillis);
            }
            catch (InterruptedException e) {
                // Check if the task is cancelled
                if (this.isCancelled()) {
                    break;
                }
            }

            // Check if the number is a prime number
            if (PrimeUtil.isPrime(i)) {
                // Add to the list
                results.add(i);

                // Publish the read-only list to give
the GUI access to the
                // partial results
                updateValue(
                    FXCollections.<Long>unmodifiableOb
servableList(
                        results));
            }
        }

        // Update the progress
        updateProgress(counter, count);
    }

    return results;
}

```

```

    }

    @Override
    protected void cancelled() {
        super.cancelled();
        updateMessage("The task was cancelled.");
    }

    @Override
    protected void failed() {
        super.failed();
        updateMessage("The task failed.");
    }

    @Override
    public void succeeded() {
        super.succeeded();
        updateMessage("The task finished successfully.");
    }
}
}

```

The program in Listing 27-7 contains the complete code to build a GUI using your PrimeFinderTask class. Figure 27-5 shows the window when the task is running. You will need to click the *Start* button to start the task. Clicking the *Cancel* button cancels the task. Once the task finishes, it is cancelled or it fails; you cannot restart it and both the *Start* and *Cancel* buttons are disabled. Notice that when the task finds a new prime number, it is displayed on the window immediately.

### ***Listing 27-7.*** Executing a Task in GUI Environment

```

// OneShotTask.java
package com.jdojo.concurrent;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;
import static javafx.concurrent.Worker.State.READY;
import static javafx.concurrent.Worker.State.RUNNING;

public class OneShotTask extends Application {
    Button startBtn = new Button("Start");
    Button cancelBtn = new Button("Cancel");
    Button exitBtn = new Button("Exit");

    // Create the task
    PrimeFinderTask task = new PrimeFinderTask();

    public static void main(String[] args) {

```

```
        Application.launch(args);
    }

    @Override
    public void start(Stage stage) {
        // Add event handlers to the buttons
        startBtn.setOnAction(e -> startTask());
        cancelBtn.setOnAction(e -> task.cancel());
        exitBtn.setOnAction(e -> stage.close());

        // Enable/Disable the Start and Cancel buttons
        startBtn.disableProperty().bind(task.stateProperty()
.isNotEqualTo(READY));
        cancelBtn.disableProperty().bind(task.stateProperty(
).isNotEqualTo(RUNNING));
        GridPane pane = new WorkerStateUI(task);
        HBox buttonBox = new HBox(5, startBtn, cancelBtn,
exitBtn);
        BorderPane root = new BorderPane();
        root.setCenter(pane);
        root.setBottom(buttonBox);
        root.setStyle("-fx-padding: 10;" +
"-fx-border-style: solid inside;" +
"-fx-border-width: 2;" +
"-fx-border-insets: 5;" +
"-fx-border-radius: 5;" +
"-fx-border-color: blue;");
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("A Prime Number Finder Task");
        stage.show();
    }

    public void startTask() {
        // Schedule the task on a background thread
        Thread backgroundThread = new Thread(task);
        backgroundThread.setDaemon(true);
        backgroundThread.start();
    }
}
```

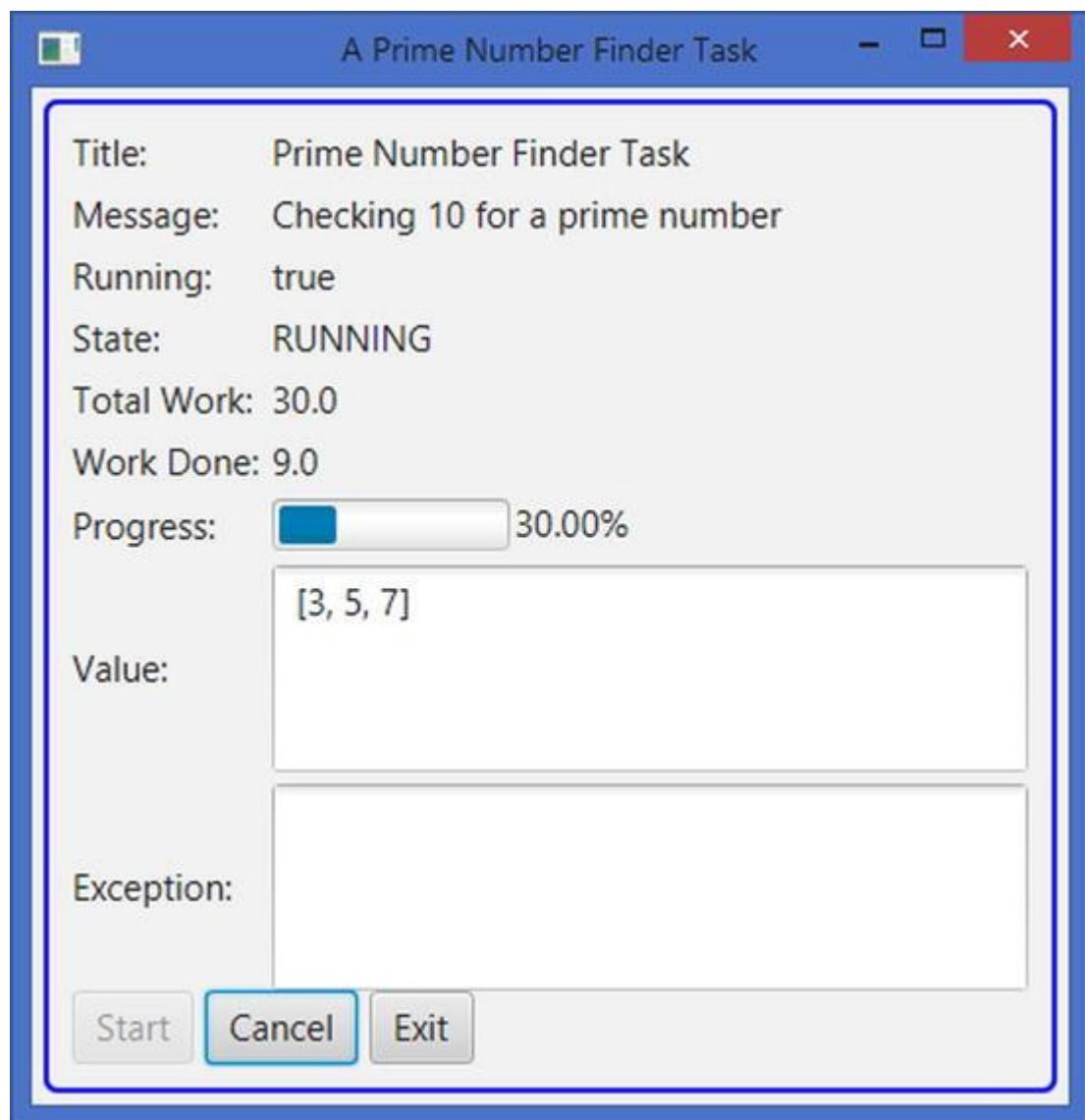


Figure 27-5. A window using the prime number finder Task

## Using the Service<V> Class

The `Service<V>` class is an implementation of the `Worker<V>` interface. It encapsulates a `Task<V>`. It makes the `Task<V>` reusable by letting it be started, cancelled, reset, and restarted.

### Creating the Service

Remember that a `Service<V>` encapsulates a `Task<V>`. Therefore, you need a `Task<V>` to have a `Service<V>`. The `Service<V>` class contains an abstract protected `createTask()` method that returns a `Task<V>`. To create a service, you need to subclass

the `Service<V>` class and provide an implementation for the `createTask()` method.

The following snippet of code creates a `Service` that encapsulates a `PrimeFinderTask`, which you have created earlier:

```
// Create a service
Service<ObservableList<Long>> service = new
Service<ObservableList<Long>>() {
    @Override
    protected Task<ObservableList<Long>> createTask() {
        // Create and return a Task
        return new PrimeFinderTask();
    }
};
```

The `createTask()` method of the service is called whenever the service is started or restarted.

## Updating Service Properties

The `Service` class contains all properties (`title`, `message`, `state`, `value`, etc.) that represent the internal state of a `Worker`. It adds an `executor` property, which is a `java.util.concurrent.Executor`. The property is used to run the `Service`. If it is not specified, a daemon thread is created to run the `Service`.

Unlike the `Task` class, the `Service` class does not contain `updateXXX()` methods for updating its properties. Its properties are bound to the corresponding properties of the underlying `Task<V>`. When the `Task` updates its properties, the changes are reflected automatically to the `Service` and to the client.

## Listening to Service Transition Events

The `Service` class contains all properties for setting the state transition listeners as contained by the `Task` class. It adds an `onReady` property. The property specifies a state transition event handler, which is called when the `Service` transitions to the `READY` state. Note that the `Task` class does not contain an `onReady` property as a `Task` is in the `READY` state when it is created and it never transitions to the `READY` state again. However, a `Service` can be in the `READY` state multiple times. A `Service` transitions to the `READY` state when it is created, reset, and restarted. The `Service` class also contains a protected `ready()` method, which is intended to be overridden by subclasses. The `ready()` method is called when the `Service` transitions to the `READY` state.

## Cancelling the Service

Use the `cancel()` methods to cancel a Service: the method sets the state of the Service to CANCELLED.

## Starting the Service

Calling the `start()` method of the Service class starts a Service. The method calls the `createTask()` method to get a Task instance and runs the Task. The Service must be in the READY state when its `start()` method is called.

```
Service<ObservableList<Long>> service = create a service  
...  
// Start the service  
service.start();
```

## Resetting the Service

Calling the `reset()` method of the Service class resets the Service. Resetting puts all the Service properties back to their initial states. The state is set to READY. Resetting a Service is allowed only when the Service is in one of the finish states: SUCCEEDED, FAILED, CANCELLED, or READY. Calling the `reset()` method throws a runtime exception if the Service is in the SCHEDULED or RUNNING state.

## Restarting the Service

Calling the `restart()` method of the Service class restarts a Service. It cancels the task if it exists, resets the service, and starts it. It calls the three methods on the Service object in sequence.

- `cancel()`
- `reset()`
- `start()`

## The Prime Finder Service Example

The program in Listing 27-8 shows how to use a Service. The Service object is created and stored as an instance variable. The Service object manages a PrimeFinderTask object, which is a Task to find prime numbers between two numbers. Four buttons are added: *Start/Restart*, *Cancel*, *Reset*, and *Exit*. The *Start* button is labeled *Restart* after the Service is started for the first time. The buttons do what their labels indicate. Buttons are disabled when they are

not applicative. Figure 27-6 shows a screenshot of the window after the *Start* button is clicked.

### ***Listing 27-8.*** Using a Service to Find Prime Numbers

```
// PrimeFinderService.java
package com.jdojo.concurrent;

import javafx.application.Application;
import javafx.application.Platform;
import javafx.beans.binding.Bindings;
import javafx.collections.ObservableList;
import javafx.concurrent.Service;
import javafx.concurrent.Task;
import static javafx.concurrent.Worker.State.RUNNING;
import static javafx.concurrent.Worker.State.SCHEDULED;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class PrimeFinderService extends Application {
    Button startBtn = new Button("Start");
    Button cancelBtn = new Button("Cancel");
    Button resetBtn = new Button("Reset");
    Button exitBtn = new Button("Exit");
    boolean onceStarted = false;

    // Create the service
    Service<ObservableList<Long>> service = new
    Service<ObservableList<Long>>() {
        @Override
        protected Task<ObservableList<Long>> createTask() {
            return new PrimeFinderTask();
        }
    };

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

    @Override
    public void start(Stage stage) {
        // Add event handlers to the buttons
        addEventHandlers();

        // Enable disable buttons based on the service state
        bindButtonsState();

        GridPane pane = new WorkerStateUI(service);
        HBox buttonBox = new HBox(5, startBtn, cancelBtn,
        resetBtn, exitBtn);
    }
}
```

```

        BorderPane root = new BorderPane();
        root.setCenter(pane);
        root.setBottom(buttonBox);
        root.setStyle("-fx-padding: 10;" +
                      "-fx-border-style: solid inside;" +
                      "-fx-border-width: 2;" +
                      "-fx-border-insets: 5;" +
                      "-fx-border-radius: 5;" +
                      "-fx-border-color: blue;");

        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("A Prime Number Finder Service");
        stage.show();
    }

    public void addEventHandlers() {
        // Add event handlers to the buttons
        startBtn.setOnAction(e -> {
            if (onceStarted) {
                service.restart();
            } else {
                service.start();
                onceStarted = true;
                startBtn.setText("Restart");
            }
        });

        cancelBtn.setOnAction(e -> service.cancel());
        resetBtn.setOnAction(e -> service.reset());
        exitBtn.setOnAction(e -> Platform.exit());
    }

    public void bindButtonsState() {
        cancelBtn.disableProperty().bind(service.stateProperty()
                                         .isNotEqualTo(RUNNING));
        resetBtn.disableProperty().bind(
            Bindings.or(service.stateProperty().isEqualTo(
                RUNNING),
                        service.stateProperty().isEqualTo(SCHEDULED)));
    }
}

```

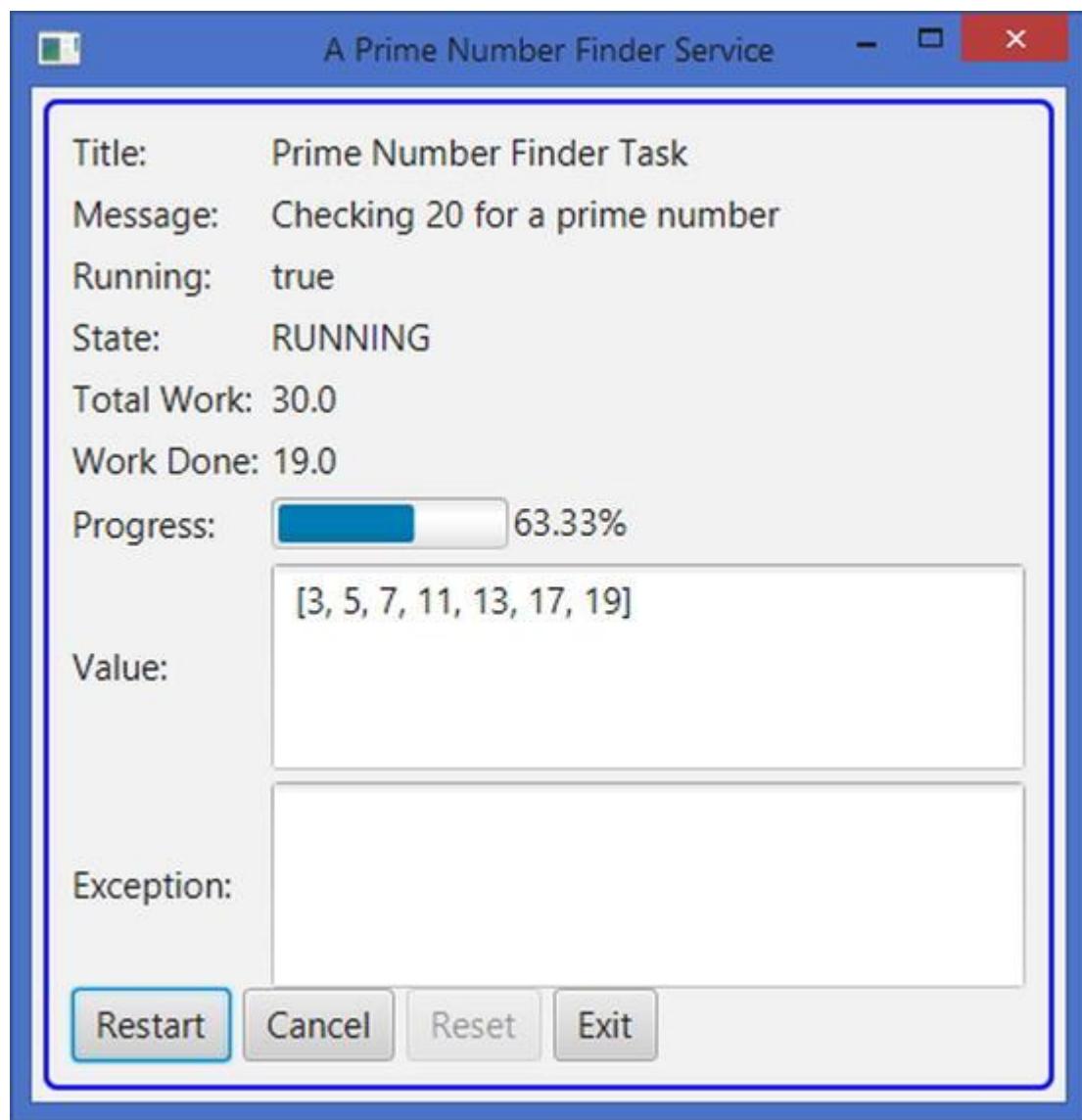


Figure 27-6. A window using a Service to find prime numbers

## Using the `ScheduledService<V>` Class

The `ScheduledService<V>` is a `Service<V>`, which automatically restarts. It can restart when it finishes successfully or when it fails. Restarting on a failure is configurable.

The `ScheduledService<V>` class inherits from the `Service<V>` class. The `ScheduledService` is suitable for tasks that use polling. For example, you may use it to refresh the score of a game or the weather report from the Internet after every 10 minutes.

### Creating the `ScheduledService`

The process of creating a `ScheduledService` is the same as that of creating a `Service`. You need to subclass

the `ScheduledService<V>` class and provide an implementation for the `createTask()` method.

The following snippet of code creates a `ScheduledService` that encapsulates a `PrimeFinderTask`, which you have created earlier:

```
// Create a scheduled service
ScheduledService<ObservableList<Long>> service =
    new ScheduledService <ObservableList<Long>>()
{
    @Override
    protected Task<ObservableList<Long>> createTask() {
        // Create and return a Task
        return new PrimeFinderTask();
    }
};
```

The `createTask()` method of the service is called when the service is started or restarted manually or automatically. Note that a `ScheduledService` is automatically restarted. You can start and restart it manually by calling the `start()` and `restart()` methods.

**Tip** Starting, cancelling, resetting, and restarting a `ScheduledService` work the same way as these operations on a `Service`.

## Updating `ScheduledService` Properties

The `ScheduledService<ScheduledService>` class inherits properties from the `Service<V>` class. It adds the following properties that can be used to configure the scheduling of the service.

- `lastValue`
- `delay`
- `period`
- `restartOnFailure`
- `maximumFailureCount`
- `backoffStrategy`
- `cumulativePeriod`
- `currentFailureCount`
- `maximumCumulativePeriod`

A `ScheduledService<V>` is designed to run several times. The current value computed by the service is not very meaningful. Your class adds a new property `lastValue`, which is of the type `V`, and it is the last value computed by the service.

The `delay` is a `Duration`, which specifies a delay between when the service is started and when it begins running. The service stays in

the SCHEDED state for the specified delay. The delay is honored only when the service is started manually calling the `start()` or `restart()` method. When the service is restarted automatically, honoring the delay property depends on the current state of the service. For example, if the service is running behind its periodic schedule, it will rerun immediately, ignoring the delay property. The default delay is zero.

The `period` is a `Duration`, which specifies the minimum amount of time between the last run and the next run. The default period is zero.

The `restartOnFailure` specifies whether the service restarts automatically when it fails. By default, it is set to true.

The `currentFailureCount` is the number of times the scheduled service has failed. It is reset to zero when the scheduled service is restarted manually.

The `maximumFailureCount` specifies the maximum number of times the service can fail before it is transitioned into the FAILED state and it is not automatically restarted again. Note that you can restart a scheduled service any time manually. By default, it is set to `Integer.MAX_VALUE`.

The `backoffStrategy` is a `Callback<ScheduledService<?>, Duration>` that computes the `Duration` to add to the period on each failure. Typically, if a service fails, you want to slow down before retrying it. Suppose a service runs every 10 minutes. If it fails for the first time, you may want to restart it after 15 minutes. If it fails for the second time, you want to increase the rerun time to 25 minutes, and so on. The `ScheduledService` class provides three built-in backoff strategies as constants.

- EXPONENTIAL\_BACKOFF\_STRATEGY
- LINEAR\_BACKOFF\_STRATEGY
- LOGARITHMIC\_BACKOFF\_STRATEGY

The rerun gaps are computed based on the non-zero period and the current failure count. The time between consecutive failed runs increases exponentially in the exponential `backoffStrategy`, linearly in the linear `backoffStrategy`, and logarithmically in the logarithmic `backoffStrategy`.

The `LOGARITHMIC_BACKOFF_STRATEGY` is the default. When the `period` is zero, the following formulas are used. The computed duration is in milliseconds.

- Exponential: `Math.exp(currentFailureCount)`

- Linear: currentFailureCount
- Logarithmic: Math.log1p(currentFailureCount)

The following formulas are used for the non-null period:

- Exponential: period + (period \* Math.exp(currentFailureCount))
- Linear: period + (period \* currentFailureCount)
- Logarithmic: period + (period \* Math.log1p(currentFailureCount))

The cumulativePeriod is a Duration, which is the time between the current failed run and the next run. Its value is computed using the backoffStrategy property. It is reset upon a successful run of the scheduled service. Its value can be capped using the maximumCumulativePeriod property.

### Listening to ScheduledService Transition Events

The ScheduledService goes through the same transition states as the Service. It goes through the READY, SCHEDULED, and RUNNING states automatically after a successful run. Depending on how the scheduled service is configured, it may go through the same state transitions automatically after a failed run.

You can listen to the state transitions and override the transition-related methods (ready(), running(), failed(), etc.) as you can for a Service. When you override the transition-related methods in a ScheduledService subclass, make sure to call the super method to keep your ScheduledService working properly.

### The Prime Finder ScheduledService Example

Let us use the PrimeFinderTask with a ScheduledService. Once started, the ScheduledService will keep rerunning forever. If it fails five times, it will quit by transitioning itself to the FAILED state. You can cancel and restart the service manually any time.

The program in Listing 27-9 shows how to use a ScheduledService. The program is very similar to the one shown in Listing 27-8, except at two places. The service is created by subclassing the ScheduledService class.

```
// Create the scheduled service
ScheduledService<ObservableList<Long>> service = new
ScheduledService<ObservableList<Long>>() {
```

```

@Override
protected Task<ObservableList<Long>> createTask() {
    return new PrimeFinderTask();
}
};

```

The ScheduledService is configured in the beginning of the start() method, setting the delay, period, and maximumFailureCount properties.

```

// Configure the scheduled service
service.setDelay(Duration.seconds(5));
service.setPeriod(Duration.seconds(30));
service.setMaximumFailureCount(5);

```

Figure 27-7, Figure 27-8, and Figure 27-9 show the state of the ScheduledService when it is not started, when it is observing the delay period in the SCHEDULED state, and when it is running. Use the Cancel and Reset buttons to cancel and reset the service. Once the service is cancelled, you can restart it manually by clicking the Restart button.

### ***Listing 27-9.*** Using a ScheduledService to Run a Task

```

// PrimeFinderScheduledService.java
package com.jdojo.concurrent;

import javafx.application.Application;
import javafx.application.Platform;
import javafx.beans.binding.Bindings;
import javafx.collections.ObservableList;
import javafx.concurrent.ScheduledService;
import javafx.concurrent.Task;
import static javafx.concurrent.Worker.State.RUNNING;
import static javafx.concurrent.Worker.State.SCHEDULED;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;
import javafx.util.Duration;

public class PrimeFinderScheduledService extends Application {
    Button startBtn = new Button("Start");
    Button cancelBtn = new Button("Cancel");
    Button resetBtn = new Button("Reset");
    Button exitBtn = new Button("Exit");
    boolean onceStarted = false;

    // Create the scheduled service
    ScheduledService<ObservableList<Long>> service =
        new ScheduledService<ObservableList<Long>>() {

```

```

    @Override
    protected Task<ObservableList<Long>> createTask() {
        return new PrimeFinderTask();
    }
}

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

@Override
public void start(Stage stage) {
    // Configure the scheduled service
    service.setDelay(Duration.seconds(5));
    service.setPeriod(Duration.seconds(30));
    service.setMaximumFailureCount(5);

    // Add event handlers to the buttons
    addEventHandlers();

    // Enable disable buttons based on the service state
    bindButtonsState();

    GridPane pane = new WorkerStateUI(service);
    HBox buttonBox = new HBox(5, startBtn, cancelBtn,
    resetBtn, exitBtn);
    BorderPane root = new BorderPane();
    root.setCenter(pane);
    root.setBottom(buttonBox);
    root.setStyle("-fx-padding: 10;" +
                  "-fx-border-style: solid inside;" +
                  "-fx-border-width: 2;" +
                  "-fx-border-insets: 5;" +
                  "-fx-border-radius: 5;" +
                  "-fx-border-color: blue;");

    Scene scene = new Scene(root);
    stage.setScene(scene);
    stage.setTitle("A Prime Number Finder Scheduled
Service");
    stage.show();
}

public void addEventHandlers() {
    // Add event handlers to the buttons
    startBtn.setOnAction(e -> {
        if (onceStarted) {
            service.restart();
        } else {
            service.start();
            onceStarted = true;
            startBtn.setText("Restart");
        }
    });
}

cancelBtn.setOnAction(e -> service.cancel());

```

```
        resetBtn.setOnAction(e -> service.reset());
        exitBtn.setOnAction(e -> Platform.exit());
    }

    public void bindButtonsState() {
        cancelBtn.disableProperty().bind(service.stateProperty()
            .isNotEqualTo(RUNNING));
        resetBtn.disableProperty().bind(
            Bindings.or(service.stateProperty().isEqualTo(
                RUNNING),
                service.stateProperty().isEqualTo(SCHEDULED)));
    }
}
```

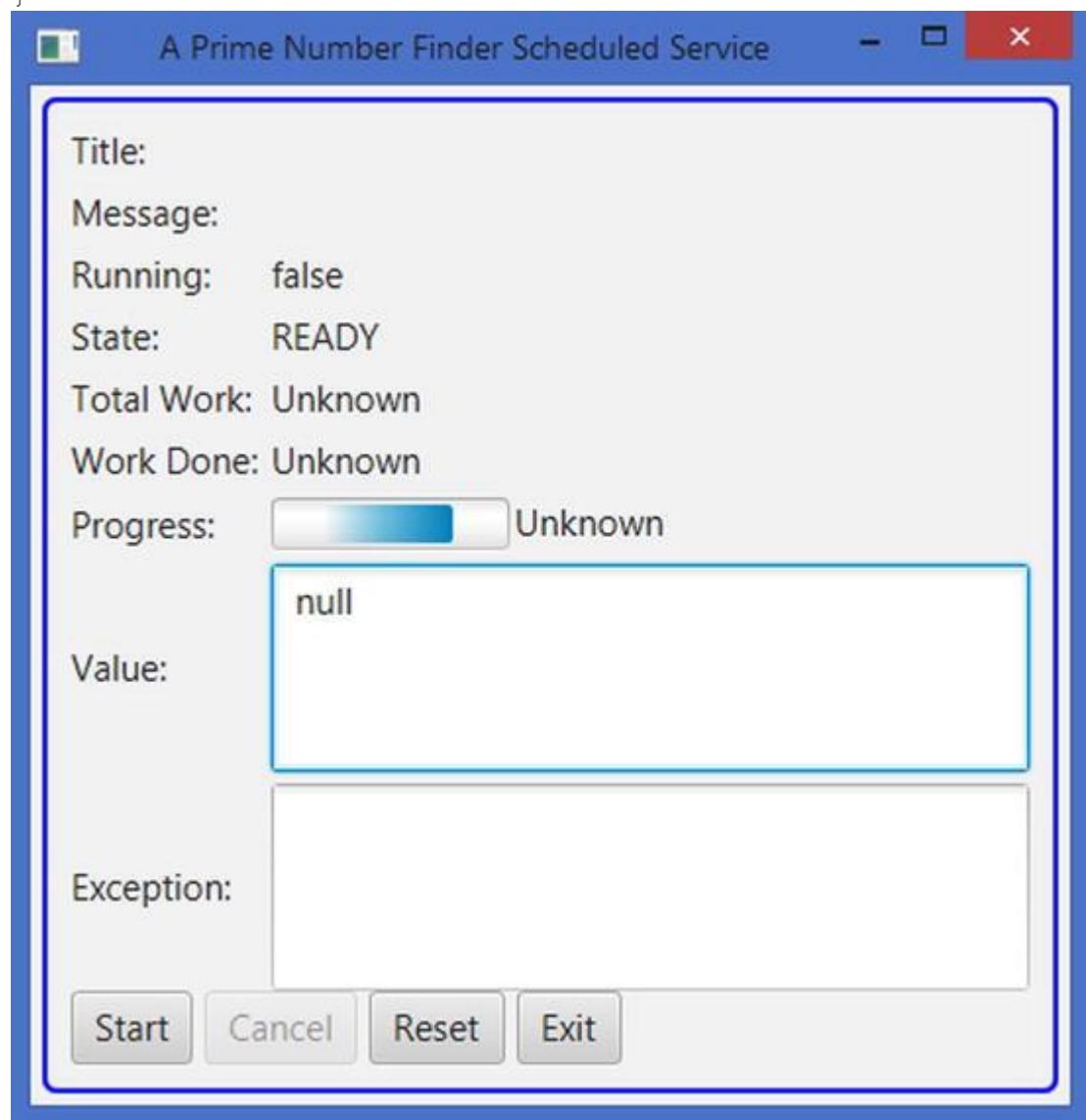
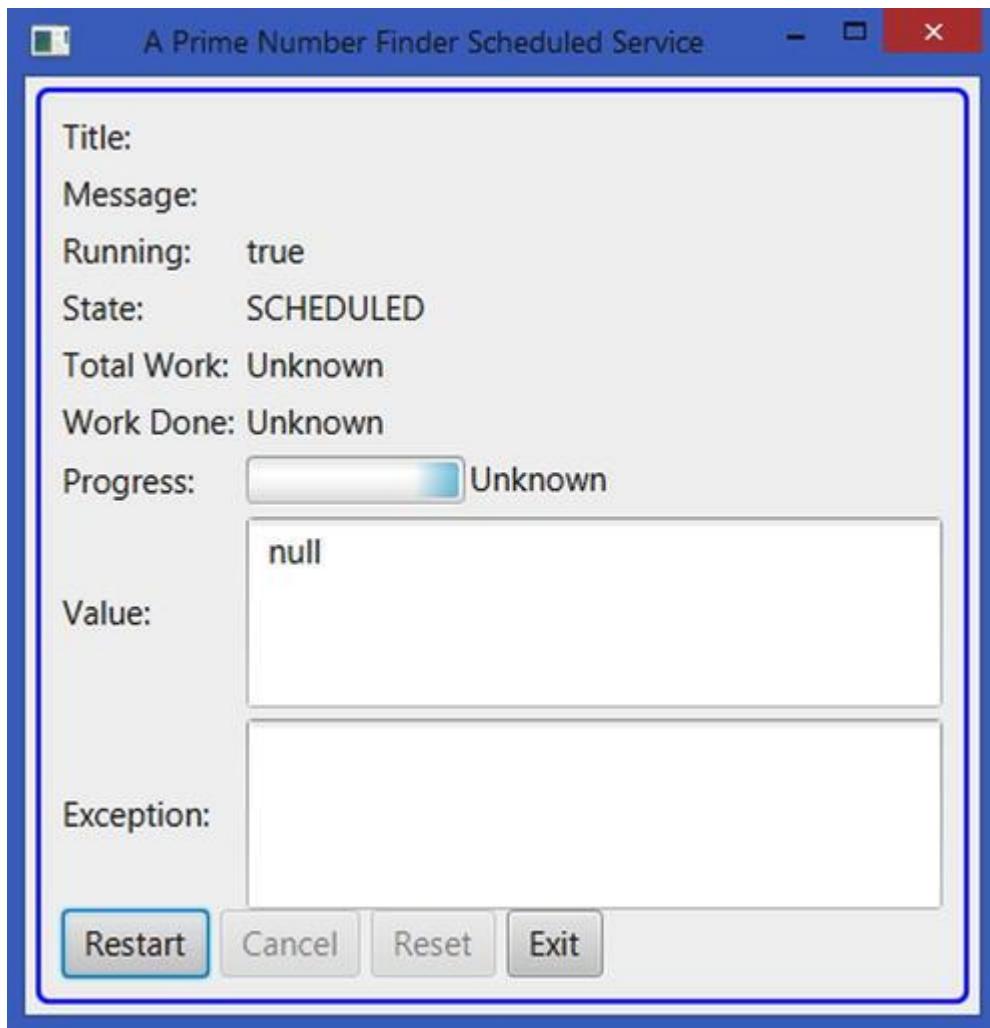


Figure 27-7. The ScheduledService is not started



**Figure 27-8.** The `ScheduledService` is started for the first time and it is observing the delay period

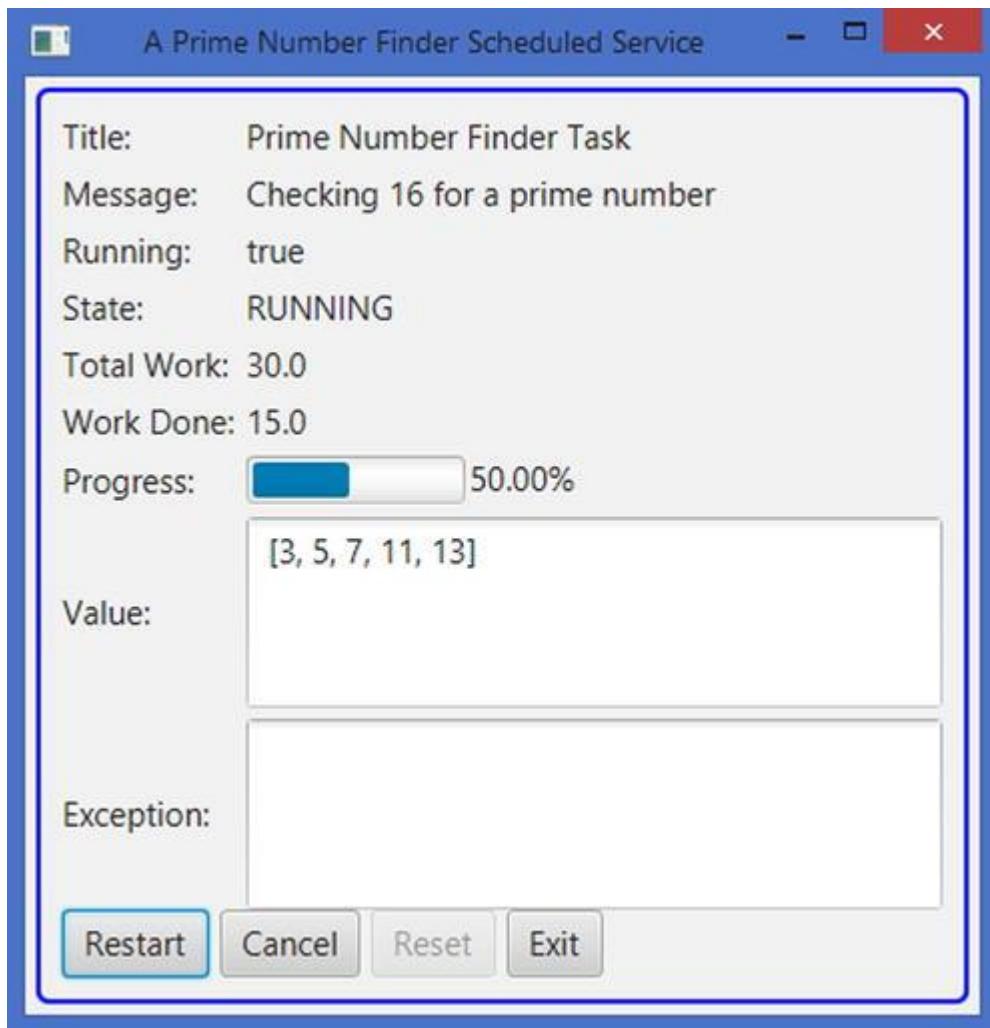


Figure 27-9. The ScheduledService is started and running

## Summary

Java (including JavaFX) GUI applications are inherently multithreaded. Multiple threads perform different tasks to keep the UI in sync with the user actions. JavaFX, like Swing and AWT, uses a single thread, called JavaFX Application Thread, to process all UI events. The nodes representing UI in a scene graph are not thread-safe. Designing nodes that are not thread-safe has advantages and disadvantages. They are faster, as no synchronization is involved. The disadvantage is that they need to be accessed from a single thread to avoid being in an illegal state. JavaFX puts a restriction that a live scene graph must be accessed from one and only one thread, the JavaFX Application Thread. This restriction indirectly imposes another restriction that a UI event should not process a long-running task, as it will make the application unresponsive. The user will get the impression that the application is hung. The JavaFX concurrency framework is built on top of the Java language concurrency framework keeping in mind that it will be used in a GUI environment.

The framework consists of one interface, four classes, and one enum. It provides a way to design a multithreaded JavaFX application that can perform long-running tasks in worker threads, keeping the UI responsive.

An instance of the `Worker` interface represents a task that needs to be performed in one or more background threads. The state of the task is observable from the JavaFX Application Thread. The `Task`, `Service`, and `ScheduledService` classes implement the `Worker` interface. They represent different types of tasks. They are abstract classes.

An instance of the `Task` class represents a one-shot task.  
A Task cannot be reused.

An instance of the `Service` class represents a reusable task.

The `ScheduledService` class inherits from the `Service` class.  
A `ScheduledService` is a task that can be scheduled to run repeatedly after a specified interval.

The constants in the `Worker.State` enum represent different states of a `Worker`. An instance of the `WorkerStateEvent` class represents an event that occurs as the state of a `Worker` changes. You can add event handlers to all three types of tasks to listen to the change in their states.

The next chapter will discuss how to incorporate audios and videos in JavaFX applications.