JavaFX

Using JavaFX UI Controls

Release 2.2

E20485-11

September 2013

In this tutorial, you learn how to build user interfaces in your JavaFX applications with the UI controls available through the JavaFX API.

ORACLE

JavaFX/Using JavaFX UI Controls, Release 2.2

E20485-11

Primary Author: Alla Redko

Contents

21 Titled Pane and Accordion

22 Menu

23 Password Field

24 Color Picker

25 Pagination Control

26 File Chooser

27 Customization of UI Controls

Part I

About This Tutorial

This tutorial covers built-in JavaFX UI controls available in the JavaFX API.

The document contains the following chapters:

- Customization of UI Controls

Each chapter provides code samples and applications to illustrate how a particular UI control functions. You can find the source files of the applications and the corresponding NetBeans projects in the table of contents.

1

JavaFX UI Controls

This chapter provides an overview of the JavaFX UI controls available through the API.

The JavaFX UI controls are built by using nodes in the scene graph. Therefore, the controls can use the visually rich features of the JavaFX platform. Because the JavaFX APIs are fully implemented in Java, you can easily integrate the JavaFX UI controls into your existing Java applications.

Figure 1–1 shows the typical UI controls you can find in the Ensemble sample application. Try this application to evaluate the wide range of controls, their behavior, and available styles.

Figure 1–1 JavaFX UI Controls

Supported UI Controls in JavaFX 2

The classes to construct UI controls reside in the `javafx.scene.control` package of the API.

The list of UI controls includes typical UI components that you might recognize from your previous development of client applications in Java. However, the JavaFX 2 SDK

introduces new Java UI controls, like `TitledPane`, `ColorPicker`, and `Pagination`.

Figure 1–2 shows a screen capture of three `TitledPane` elements with lists of settings for a social network application. The lists can slide in (retract) and slide out (extend).

Figure 1–2 Titled Panes

See the API documentation for the complete list of UI controls.

The UI control classes provide additional variables and methods beyond those of the `Control` class to support typical user interactions in an intuitive manner. You can assign a specific style to your UI components by applying Cascading Style Sheets (CSS). For some unusual tasks, you might need to extend the `Control` class to create a custom UI component, or use the `Skin` interface to define a new skin for an existing control.

Features and Effects

Because UI controls from the `javafx.scene.control` package are all extensions of the `Node` class, they can be integrated with the scene graph rendering, animation, transformations, and animated transitions.

Consider the task of creating a button, applying a reflection to it, and animating the button by altering its opacity from its maximum value to its minimum value.

Figure 1–3 shows three states of the button through the animation timeline. The left image shows the button when its opacity is set to `1.0`, the central image shows the opacity set to `0.8`, and the right image shows the opacity set to `0.5`.

Figure 1–3 Animated Button

By using the JavaFX APIs, you can implement this task with only a few lines of code.

Example 1–1 creates and starts an indefinite timeline, where within a key frame of 600 milliseconds the button's opacity changes from its default value (1.0) to 0.0. The `setAutoReverse` method enables the reverse order.

Example 1–1 Creating an Animated Button

```
import javafx.animation.KeyFrame;
import javafx.animation.KeyValue;
import javafx.animation.Timeline;
import javafx.util.Duration;
```

```
import javafx.scene.control.Button;
import javafx.scene.text.Font;
import javafx.scene.effect.Reflection;

...
Button button = new Button();
    button.setText("OK");
    button.setFont(new Font("Tahoma", 24));
    button.setEffect(new Reflection());

final Timeline timeline = new Timeline();
timeline.setCycleCount(Timeline.INDEFINITE);
timeline.setAutoReverse(true);
final KeyValue kv = new KeyValue(button.opacityProperty(), 0);
final KeyFrame kf = new KeyFrame(Duration.millis(600), kv);
timeline.getKeyFrames().add(kf);
timeline.play();
...
```

You can also apply other visual effects that are available in the `javafx.scene.effect` package, such as shadow, lighting, or motion blur.

Styling UI Controls with CSS

You can customize the look of the built-in UI controls by defining your own Cascading Style Sheets (CSS). Using CSS in JavaFX applications is much the same as using CSS in HTML, because each case is based on the same CSS specification. The visual state of a control is defined by the .css file, as shown in Example 1–2.

Example 1–2 Defining Styles for UI Controls in the CSS File

```
/*controlStyle.css */

.scene{
    -fx-font: 14pt "Cambria Bold";
    -fx-color: #e79423;
    -fx-background: #67644e;
}

.button{
    -fx-text-fill: #006464;
    -fx-background-color: #e79423;
    -fx-border-radius: 20;
    -fx-background-radius: 20;
    -fx-padding: 5;
}
```

You can enable the style in the application through the `getStylesheets` method of the `Scene` class, as shown in Example 1–3.

Example 1–3 Applying CSS

```
Scene scene = new Scene();
scene.getStylesheets().add("uicontrolssample/controlStyle.css");
```

Additionally, you define the style of a control directly in the code of your application by using the `setStyle` method. The `-fx-base` property defined for the toggle button in Example 1–4 overrides the corresponding properties defined in the .css file for all the controls added to the scene.

Example 1–4 Defining the Style of a Toggle Button in the JavaFX Application

```
ToggleButton tb3 = new ToggleButton ("I don't know");
tb3.setStyle("-fx-base: #ed1c24;");
```

Figure 1–4 shows how the styled toggle button looks when it is added to the application.

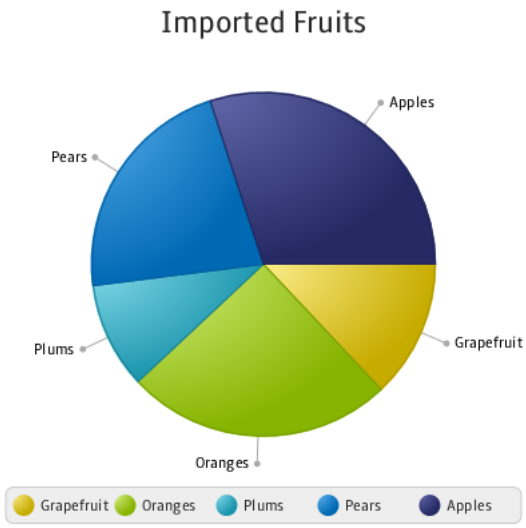
Figure 1–4 Applying CSS Style to a Toggle Button

Charts

In addition to the typical elements of a user interface, JavaFX SDK provides prefabricated charts in the `javafx.scene.chart` package. The following types of charts are currently supported: area chart, bar chart, bubble chart, line chart, pie chart, and scatter chart. A chart might contain several series of data.

Figure 1–5 shows a pie chart of imported fruits.

Figure 1–5 Pie Chart



Unlike with other Java client toolkits, with the JavaFX SDK you can build such a chart in your application by adding just a few lines of code. You can also define various color schemes and styles, apply visual effects, process mouse events, and create animation.

See Using JavaFX Charts for more information about chart features and capabilities.

Integrating JavaFX 2 UI Controls in Swing

You can integrate JavaFX UI controls into existing Java client applications built on the Swing toolkit.

To integrate JavaFX content into a Swing application, use the following steps:

1. Add all the JavaFX UI controls to the `javafx.scene.Scene` object one by one, within a layout container, or as a group.
2. Add the `Scene` object to the content of the Swing application.

If you need to place a single JavaFX 2 control in your existing Swing code, you must perform both of the preceding steps.

Even when they are integrated into a Swing application, the JavaFX 2 UI controls are still rendered by using the Prism graphical library and take full advantage of its advanced rendering capabilities.

See the JavaFX in Swing tutorial for more information about JavaFX and Swing interoperability.

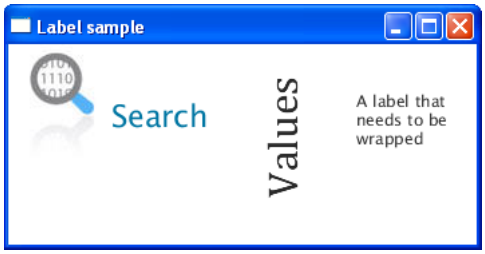
2

Label

This chapter explains how to use the `Label` class that resides in the `javafx.scene.control` package of the JavaFX API to display a text element. Learn how to wrap a text element to fit the specific space, add a graphical image, or apply visual effects.

Figure 2–1 shows three common label usages. The label at the left is a text element with an image, the label in the center represents rotated text, and the label at the right renders wrapped text.

Figure 2–1 Sample Application with Labels



Creating a Label

The JavaFX API provides three constructors of the `Label` class for creating labels in your application, as shown in Example 2–1.

Example 2–1 Creating Labels

```
//An empty label
Label label1 = new Label();
//A label with the text element
Label label2 = new Label("Search");
//A label with the text element and graphical icon
Image image = new Image(getClass().getResourceAsStream("labels.jpg"));
Label label3 = new Label("Search", new ImageView(image));
```

Once you have created a label in your code, you can add textual and graphical content to it by using the following methods of the `Labeled` class.

- The `setText(String text)` method – specifies the text caption for the label
- `setGraphic(Node graphic)` – specifies the graphical icon

The `setTextFill` method specifies the color to paint the text element of the label. Study Example 2–2. It creates a text label, adds an icon to it, and specifies a fill color for the text.

Example 2–2 Adding an Icon and Text Fill to a Label

```
Label label1 = new Label("Search");
Image image = new Image(getClass().getResourceAsStream("labels.jpg"));
label1.setGraphic(new ImageView(image));
label1.setTextFill(Color.web("#0076a3"));
```

When this code fragment is added to the application, it produces the label shown in Figure 2–2.

Figure 2–2 Label with Icon

When defining both text and graphical content for your button, you can use the `setGraphicTextGap` method to set the gap between them.

Additionally, you can vary the position of the label content within its layout area by using the `setTextAlignment` method. You can also define the position of the graphic relative to the text by applying the `setContentDisplay` method and specifying one of the following `ContentDisplay` constant: `LFFT`, `RIGHT`, `CENTER`, `TOP`, `BOTTOM`.

Setting a Font

Compare the Search labels in Figure 2–1 and Figure 2–2. Notice that the label in Figure 2–1 has a larger font size. This is because the code fragment shown in Example 2–2 does not specify any font settings for the label. It is rendered with the default font size.

To provide a font text size other than the default for your label use the `setFont` method of the `Labeled` class. The code fragment in Example 2–3 sets the size of the label1 text to 30 points and the font name to Arial. For label2 sets the text size to 32 points and the font name to Cambria.

Example 2–3 Applying Font Settings

```
//Use a constructor of the Font class
label1.setFont(new Font("Arial", 30));
//Use the font method of the Font class
label2.setFont(Font.font("Cambria", 32));
```

Wrapping Text

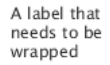
When you create a label, sometimes you must fit it within a space that is smaller than you need to render. To break up (wrap) the text so that is can fit into the layout area, set the `true` value for the `setWrapText` method, as shown in Example 2–4.

Example 2–4 Enable Text Wrapping

```
Label label3 = new Label("A label that needs to be wrapped");
label3.setWrapText(true);
```

When label3 is added to the content of an application, it is rendered as shown in Figure 2–3.

Figure 2–3 Label with Wrapped Text



Suppose that the layout area of the label is limited not only by its width, but also by its height. You can specify the behavior of a label when it is impossible to render the entire required text string. Use the `setTextOverrun` method of the `Labeled` class and one of the available `OverrunStyle` types to define how to process the part of the text string that cannot be rendered properly. See the API documentation for more information about `OverrunStyle` types.

Applying Effects

Although a label is static content and cannot be edited, you can apply visual effects or transformations to it. The code fragment in Example 2–5 rotates label2 270 degrees and translates its position vertically.

Example 2–5 Rotating a Label

```
Label label2 = new Label ("Values");
label2.setFont(new Font("Cambria", 32));
label2.setRotate(270);
label2.setTranslateY(50);
```

Rotation and translation are typical transformations available in the JavaFX API. Additionally, you can set up an effect that zooms (magnifies) the label when a user hovers the mouse cursor over it.

The code fragment shown in Example 2–6 applies the zoom effect to label3. When the `MOUSE_ENTERED` event is fired on the label, the scaling factor of 1.5 is set for the `setScaleX` and `setScaleY` methods. When a user moves the mouse cursor off of the label and the `MOUSE_EXITED` event occurs, the scale factor is set to 1.0, and the label is rendered in its original size.

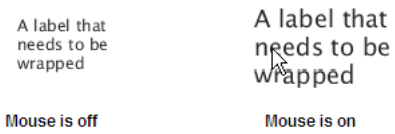
Example 2–6 Applying the Zoom Effect

```
label3.setOnMouseEntered(new EventHandler<MouseEvent>() {
    @Override public void handle(MouseEvent e) {
        label3.setScaleX(1.5);
        label3.setScaleY(1.5);
    }
});

label3.setOnMouseExited(new EventHandler<MouseEvent>() {
    @Override public void handle(MouseEvent e) {
        label3.setScaleX(1);
        label3.setScaleY(1);
    }
```

```
});
```

Figure 2–4 shows the two states of label3.

Figure 2–4 Zooming a Label



Related API Documentation

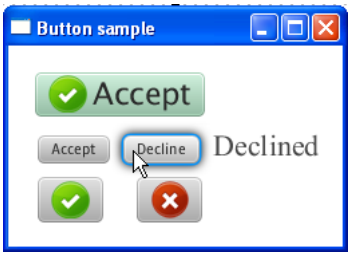
- `Label`
- `Labeled`

3

Button

The `Button` class available through the JavaFX API enables developers to process an action when a user clicks a button. The `Button` class is an extension of the `Labeled` class. It can display text, an image, or both. Figure 3–1 shows buttons with various effects. In this chapter you will learn how to create each of these button types.

Figure 3–1 Types of Buttons



Creating a Button

You can create a `Button` control in a JavaFX application by using three constructors of the Button class as shown on Example 3–1.

Example 3–1 Creating a Button

```
//A button with an empty text caption.
Button button1 = new Button();
//A button with the specified text caption.
Button button2 = new Button("Accept");
//A button with the specified text caption and icon.
Image imageOk = new Image(getClass().getResourceAsStream("ok.png"));
Button button3 = new Button("Accept", new ImageView(imageOk));
```

Because the `Button` class extends the `Labeled` class, you can use the following methods to specify content for a button that does not have an icon or text caption:

- The `setText(String text)` method – specifies the text caption for the button
- The `setGraphic(Node graphic)` method – specifies the graphical icon

Example 3–2 shows how to create a button with an icon but without a text caption.

Example 3–2 Adding an Icon to a Button

```
Image imageDecline = new Image(getClass().getResourceAsStream("not.png"));
Button button5 = new Button();
```

```
button5.setGraphic(new ImageView(imageDecline));
```

When added to the application, this code fragment produces the button shown in Figure 3–2.

Figure 3–2 Button with Icon

In Example 3–2 and Figure 3–2, the icon is an `ImageView` object. However, you can use other graphical objects, for example, shapes that reside in the `javafx.scene.shape` package. When defining both text and graphical content for your button, you can use the `setGraphicTextGap` method to set the gap between them.

The default skin of the `Button` class distinguishes the following visual states of the button. Figure 3–3 shows the default states of a button with an icon.

Figure 3–3 Button States

Assigning an Action

The primary function of each button is to produce an action when it is clicked. Use the `setOnAction` method of the `Button` class to define what will happen when a user clicks the button. Example 3–3 shows a code fragment that defines an action for button2.

Example 3–3 Defining an Action for a Button

```
button2.setOnAction(new EventHandler<ActionEvent>() {
    @Override public void handle(ActionEvent e) {
        label.setText("Accepted");
    }
});
```

`ActionEvent` is an event type that is processed by `EventHandler`. An `EventHandler` object provides the `handle` method to process an action fired for a button. Example 3–3 shows how to override the `handle` method, so that when a user presses button2 the text caption for a label is set to "Accepted."

You can use the `Button` class to set as many event-handling methods as you need to cause the specific behavior or apply visual effects.

Applying Effects

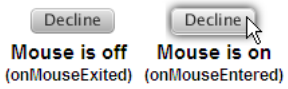
Because the `Button` class extends the `Node` class, you can apply any of the effects in the `javafx.scene.effect` package to enhance the visual appearance of the button. In Example 3–4, the `DropShadow` effect is applied to button3 when the `onMouseEntered` event occurs.

Example 3–4 Applying the DropShadow Effect

```
DropShadow shadow = new DropShadow();
//Adding the shadow when the mouse cursor is on
button3.addEventHandler(MouseEvent.MOUSE_ENTERED,
    new EventHandler<MouseEvent>() {
        @Override public void handle(MouseEvent e) {
            button3.setEffect(shadow);
        }
});
//Removing the shadow when the mouse cursor is off
button3.addEventHandler(MouseEvent.MOUSE_EXITED,
    new EventHandler<MouseEvent>() {
        @Override public void handle(MouseEvent e) {
            button3.setEffect(null);
        }
});
```

Figure 3–4 shows the states of button3 when the mouse cursor is on it and when it is off.

Figure 3–4 Button with Drop Shadow



Styling a Button

The next step to enhance the visual appearance of a button is to apply CSS styles defined by the `Skin` class. Using CSS in JavaFX 2 applications is similar to using CSS in HTML, because each case is based on the same CSS specification.

You can define styles in a separate CSS file and enable them in the application by using the `setStyleClass` method. This method is inherited from the `Node` class and is available for all UI controls. Alternatively, you can define the style of a button directly in the code by using the `setStyle` method. Example 3–5 and Figure 3–5 demonstrate the latter approach.

Example 3–5 Styling a Button

```
button1.setStyle("-fx-font: 22 arial; -fx-base: #b6e7c9;");
```

The `-fx-font-size` property sets a font size for button1. The `-fx-base` property overrides the default color applied to the button. As the result, button1 is light green with lager text size, as shown in Figure 3–5.

Figure 3–5 Button Styled with CSS

Related API Documentation

- `Button`
- `Labeled`

4

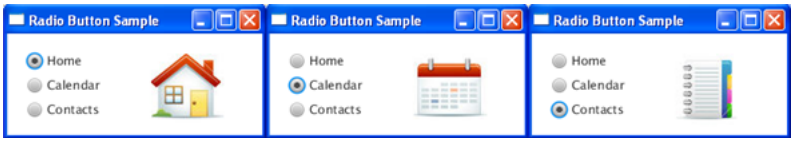
Radio Button

This chapter discusses the radio button control and the `RadioButton` class, a specialized implementation of the `ToggleButton` class.

A radio button control can be either selected or deselected. Typically radio buttons are combined into a group where only one button at a time can be selected. This behavior distinguishes them from toggle buttons, because all toggle buttons in a group can be in a deselected state.

Figure 4–1 shows three screen captures of the `RadioButton` sample, in which three radio buttons are added to a group.

Figure 4–1 RadioButton Sample



Study the following paragraphs to learn more about how to implement radio buttons in your applications.

Creating a Radio Button

The `RadioButton` class available in the `javafx.scene.control` package of the JavaFX SDK provides two constructors with which you can create a radio button. Example 4–1 shows two radio buttons. The constructor with no parameters is used to create rb1. The text caption for this radio button is set by using the `setText` method. The text caption for rb2 is defined within the corresponding constructor.

Example 4–1 Creating Radio Buttons

```
//A radio button with an empty string for its label
RadioButton rb1 = new RadioButton();
//Setting a text label
rb1.setText("Home");
//A radio button with the specified label
RadioButton rb2 = new RadioButton("Calendar");
```

You can explicitly make a radio button selected by using the `setSelected` method and specifying its value as `true`. If you need to check whether a particular radio button was selected by a user, apply the `isSelected` method.

Because the `RadioButton` class is an extension of the `Labeled` class, you can specify not only a text caption, but also an image. Use the `setGraphic` method to specify an image. Example 4–2 demonstrates how to implement a graphical radio button in your application.

Example 4–2 Creating a Graphical Radio Button

```
Image image = new Image(getClass().getResourceAsStream("ok.jpg"));
RadioButton rb = new RadioButton("Agree");
rb.setGraphic(new ImageView(image));
```

Adding Radio Buttons to Groups

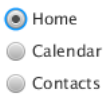
Radio buttons are typically used in a group to present several mutually exclusive options. The `ToggleGroup` object provides references to all radio buttons that are associated with it and manages them so that only one of the radio buttons can be selected at a time. Example 4–3 creates a toggle group, creates three radio buttons, adds each radio button to the toggle group, and specifies which button should be selected when the application starts.

Example 4–3 Creating a Group of Radio Buttons

```
final ToggleGroup group = new ToggleGroup();

RadioButton rb1 = new RadioButton("Home");
rb1.setToggleGroup(group);
rb1.setSelected(true);

RadioButton rb2 = new RadioButton("Calendar");
rb2.setToggleGroup(group);

RadioButton rb3 = new RadioButton("Contacts");
rb3.setToggleGroup(group);
```

When these radio buttons are laid out by using the layout containers and added to the content of the application, the output should resemble Figure 4–2.

Figure 4–2 Three Radio Buttons Combined in a Group



Processing Events for Radio Buttons

Typically, the application performs an action when one of the radio buttons in the group is selected. Review the code fragment in Example 4–4 to learn how to change an icon according to which radio button is selected.

Example 4–4 Processing Action for Radio Buttons

```
ImageView image = new ImageView();

rb1.setUserData("Home")
rb2.setUserData("Calendar");
```

```
rb3.setUserData("Contacts");

final ToggleGroup group = new ToggleGroup();
group.selectedToggleProperty().addListener(new ChangeListener<Toggle>(){
    public void changed(ObservableValue<? extends Toggle> ov,
        Toggle old_toggle, Toggle new_toggle) {
            if (group.getSelectedToggle() != null) {
                final Image image = new Image(
                    getClass().getResourceAsStream(
                        group.getSelectedToggle().getUserData().toString() +
                            ".jpg"
                    )
                );
                icon.setImage(image);
            }
        }
});
```

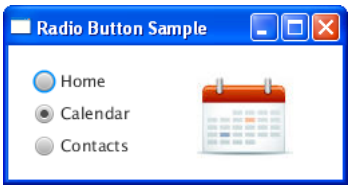
The user data was assigned for each radio button. The `ChangeListener<Toggle>` object checks a selected toggle in the group. It uses the `getSelectedToggle` method to identify which radio button is currently selected and extracts its user data by calling the `getUserData` method. Then the user data is applied to construct an image file name to load.

For example, when rb3 is selected, the `getSelectedToggle` method returns "rb3," and the `getUserData` method returns "Contacts." Therefore, the `getResourceAsStream` method receives the value "Contacts.jpg." The application output is shown in Figure 4–1.

Requesting Focus for a Radio Button

In the group of radio buttons, the first button initially has the focus by default. If you apply the `setSelected` method to the second radio button in the group, you should expect the result shown in Figure 4–3.

Figure 4–3 Default Focus Settings



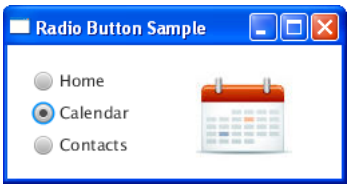
The second radio button is selected, and the first button remains in focus. Use the `requestFocus` function to change the focus, as shown in Example 4–5.

Example 4–5 Requesting Focus for the Second Radio Button

```
rb2.setSelected(true);
rb2.requestFocus();
```

When applied, this code produces the result shown in Figure 4–4.

Figure 4–4 Setting Focus for the Selected Radio Button



Related API Documentation

- `RadioButton`
- `Labeled`
- `ToggleGroup`

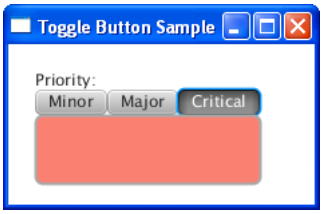
5

Toggle Button

In this chapter, you learn about the `ToggleButton` class, another type of buttons available through the JavaFX API.

Two or more toggle buttons can be combined into a group where only one button at a time can be selected, or where no selection is required. Figure 5–1 is a screen capture of an application that combines three toggle buttons in a group. The application paints the rectangle with a specific color according to on which toggle button is selected.

Figure 5–1 Three Toggle Buttons



Creating a Toggle Button

You can create a toggle button in your application by using any of the three constructors of the `ToggleButton` class, as shown in Example 5–1.

Example 5–1 Creating Toggle Buttons

```
//A toggle button without any caption or icon
ToggleButton tb1 = new ToggleButton();
//A toggle button with a text caption
ToggleButton tb2 = new ToggleButton("Press me");
//A toggle button with a text caption and an icon
Image image = new Image(getClass().getResourceAsStream("icon.png"));
ToggleButton tb3 = new ToggleButton ("Press me", new ImageView(image));
```

The `ToggleButton` class is an extension of the `Labeled` class, so you can specify a text caption, an image, or both image and text. You can use the `setText` and `setGraphic` methods of the `Labeled` class to specify textual and graphical content for a toggle button.

Once you have defined toggle buttons in your code, you can combine them in a group and set a specific behavior.

Adding Toggle Buttons to a Group

The implementation of the `ToggleButton` class is very similar to the implementation the `RadioButton` class. However, unlike radio buttons, toggle buttons in a toggle group do not attempt to force the selection at least one button in the group. That is, clicking the selected toggle button causes it to become deselected, clicking the selected radio button in the group has no effect.

Take a moment to study the code fragment Example 5–2.

Example 5–2 Combining Toggle Buttons in a Group

```
final ToggleGroup group = new ToggleGroup();

ToggleButton tb1 = new ToggleButton("Minor");
tb1.setToggleGroup(group);
tb1.setSelected(true);

ToggleButton tb2 = new ToggleButton("Major");
tb2.setToggleGroup(group);

ToggleButton tb3 = new ToggleButton("Critical");
tb3.setToggleGroup(group);
```

Example 5–2 creates three toggle buttons and adds them to the toggle group. The `setSelected` method is called for the tb1 toggle button so that it is selected when the application starts. However, you can deselect the Minor toggle button so that no toggle buttons are selected in the group at startup, as shown in Figure 5–2.

Figure 5–2 Three Toggle Buttons in a Group

Typically, you use a group of toggle buttons to assign a specific behavior for each button. The next section explains how to use these toggle buttons to alter the color of a rectangle.

Setting the Behavior

The `setUserData` method inherited by the `ToggleButton` class from the `Node` class helps you to associate any selected option with a particular value. In Example 5–3, the user data indicates which color should be used to paint the rectangle.

Example 5–3 Setting User Data for the Toggle Buttons

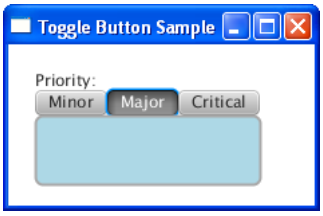
```
tb1.setUserData(Color.LIGHTGREEN);
tb2.setUserData(Color.LIGHTBLUE);
tb3.setUserData(Color.SALMON);

final Rectangle rect = new Rectangle(145, 50);

final ToggleGroup group = new ToggleGroup();
group.selectedToggleProperty().addListener(new ChangeListener<Toggle>(){
    public void changed(ObservableValue<? extends Toggle> ov,
        Toggle toggle, Toggle new_toggle) {
```

```
            if (new_toggle == null)
                rect.setFill(Color.WHITE);
            else
                rect.setFill(
                    (Color) group.getSelectedToggle().getUserData()
                );
        }
});
```

The `ChangeListener<Toggle>` object checks a selected toggle in the group. If none of the toggle buttons is selected, the rectangle is painted with the white color. If one of the toggle button is selected, consecutive calls of the `getSelectedToggle` and `getUserData` methods return a color to paint the rectangle.

For example, if a user selects the tb2 toggle button, the `setSelectedToggle().getUserData()` call returns `Color.LIGHTBLUE`. The result is shown in Figure 5–3.

Figure 5–3 Using Toggle Buttons to Paint a Rectangle



See the ToggleButtonSample.java file to examine the complete code of the application.

Styling Toggle Buttons

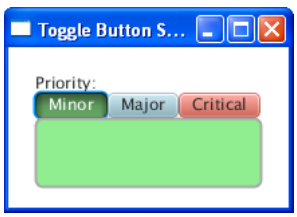
You can enhance this application by applying CSS styles to the toggle buttons. Using CSS in JavaFX 2 applications is similar to using CSS in HTML, because each case is based on the same CSS specification. Example 5–4 uses the `setStyle` method to alter the `-fx-base` CSS properties of the toggle buttons.

Example 5–4 Applying CSS Styles to Toggle Buttons

```
tb1.setStyle("-fx-base: lightgreen;");
tb2.setStyle("-fx-base: lightblue;");
tb3.setStyle("-fx-base: salmon;");
```

When added to the application code these lines change the visual appearance of the toggle buttons as shown in Figure 5–4.

Figure 5–4 Painted Toggle Buttons



You might want to try other CSS properties of the `ToggleButton` class or apply animation, transformations, and visual effects available in the JavaFX API.

Related API Documentation

- `ToggleButton`
- `ToggleGroup`

6

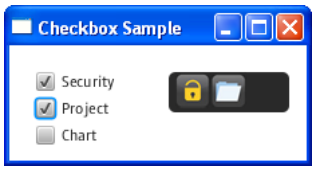
Checkbox

This chapter teaches how to add checkboxes to your JavaFX applications.

Although checkboxes look similar to radio buttons, they cannot be combined into toggle groups to enable the selection of many options at one time. See the Radio Button and Toggle Button chapters for more information.

Figure 6–1 shows a screen capture of an application in which three checkboxes are used to enable or disable icons in an application toolbar.

Figure 6–1 Checkbox Sample



Creating Checkboxes

Example 6–1 creates two simple checkboxes.

Example 6–1 Creating Checkboxes

```
//A checkbox without a caption
CheckBox cb1 = new CheckBox();
//A checkbox with a string caption
CheckBox cb2 = new CheckBox("Second");

cb1.setText("First");
cb1.setSelected(true);
```

Once you have created a checkbox, you can modify it by using methods available through the JavaFX APIs. In Example 6–1 the `setText` method defines the text caption of the c1 checkbox. The `setSelected` method is set to `true` so that the cb1 checkbox is selected when the application is started.

Defining a State

The checkbox can be either defined or undefined. When it is defined, you can select or deselect it. However, when the checkbox is undefined, it cannot be selected or deselected. Use a combination of the `setSelected` and `setIndeterminate`

methods of the `CheckBox` class to specify the state of the checkbox. Table 6–1 shows three states of a checkbox based on its `INDETERMINATE` and `SELECTED` properties.

Table 6–1 States of a Checkbox

Property Values	Checkbox Appearance
`INDETERMINATE` = false `SELECTED` = false	
`INDETERMINATE` =false `SELECTED` = true	
`INDETERMINATE` = true `SELECTED` = true/false	

You might need enabling three states for checkboxes in your application when they represent UI elements that can be in mixed states, for example, "Yes", "No", "Not Applicable." The `allowIndeterminate` property of the `CheckBox` object determines whether the checkbox should cycle through all three states: selected, deselected, and undefined. If the variable is `true`, the control will cycle through all the three states. If it is `false`, the control will cycle through the selected and deselected states. The application described in the next section constructs three checkboxes and enables only two states for them.

Setting the Behavior

The code fragment in Example 6–2 creates three checkboxes, such that if a checkbox is selected, the corresponding icon appears in a toolbar.

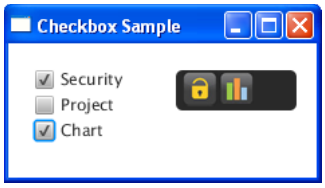
Example 6–2 Setting the Behavior for the Checkboxes

```
final String[] names = new String[]{"Security", "Project", "Chart"};
final Image[] images = new Image[names.length];
final ImageView[] icons = new ImageView[names.length];
final CheckBox[] cbs = new CheckBox[names.length];

for (int i = 0; i < names.length; i++) {
    final Image image = images[i] =
        new Image(getClass().getResourceAsStream(names[i] + ".png"));
    final ImageView icon = icons[i] = new ImageView();
    final CheckBox cb = cbs[i] = new CheckBox(names[i]);
    cb.selectedProperty().addListener(new ChangeListener<Boolean>() {
        public void changed(ObservableValue<? extends Boolean> ov,
            Boolean old_val, Boolean new_val) {
                icon.setImage(new_val ? image : null);
        }
    });
}
```

The `names` array uses a `for` loop to create an array of checkboxes and a corresponding array of icons. For example, cbs[0], the first checkbox, is assigned the "Security" text caption. At the same time, image[0] receives "Security.png" as a file name for the `getResourceStream` method when an image for the first icon is created. If a particular checkbox is selected, the corresponding image is assigned to the icon. If a checkbox is deselected, the icon receives a `null` image and the icon is not rendered.

Figure 6–2 shows an application when the Security and Chart checkboxes are selected and the Project checkbox is deselected.

Figure 6–2 Checkbox Application in Action



Styling a Checkbox

The checkboxes in Figure 6–2 have the default look and feel of the `CheckBox` class. You can alter the appearance of a checkbox by using the `setStyle` method, as shown in Example 6–3.

Example 6–3 Styling a Checkbox

```
cb1.setStyle(
    "-fx-border-color: lightblue; "
    + "-fx-font-size: 20;"
    + "-fx-border-insets: -5; "
    + "-fx-border-radius: 5;"
    + "-fx-border-style: dotted;"
    + "-fx-border-width: 2;"
);
```

The new style includes a dotted light blue border and an increased font size for its text caption. Figure 6–3 shows the cb1 checkbox with this style applied.

Figure 6–3 Styled Checkbox

To set a specific style for all the checkboxes in your application, use the following procedure:

- Create a .css file.
- Create the `checkbox` CSS class in the .css file.
- Define all the required styles in the `checkbox` CSS class.
- In your JavaFX application, enable the style sheet by using the `setStyleClass` method.

Related API Documentation

- `CheckBox`
- JavaFX CSS Specification

7

Choice Box

This chapter describes choice boxes, the UI controls that provide support for quickly selecting between a few options.

Use the `ChoiceBox` class to add choice boxes to your JavaFX applications. Its simple implementation is shown in Figure 7–1.

Figure 7–1 Creating a Choice Box with Three Items

Creating a Choice Box

Example 7–1 creates a choice box with three items.

Example 7–1 Creating a Choice Box

```
ChoiceBox cb = new ChoiceBox(FXCollections.observableArrayList(
    "First", "Second", "Third")
);
```

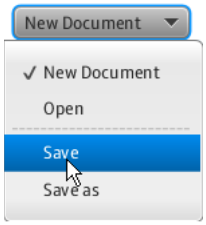
Example 7–1 shows a list of items created and populated within a constructor of the `ChoiceBox` class. The list items are specified by using an observable array. Alternatively, you can use an empty constructor of the class and set the list items by using the `setItems` method shown in Example 7–2.

Example 7–2 Choice Box with Text Elements and a Separator

```
ChoiceBox cb = new ChoiceBox();
cb.setItems(FXCollections.observableArrayList(
    "New Document", "Open ",
    new Separator(), "Save", "Save as")
);
```

Note that a choice box can contain not only text elements, but other objects as well. A `Separator` control is used in Example 7–2 to separate the items. When this code fragment is integrated into the application, it produces the output shown in Figure 7–2.

Figure 7–2 Menu Created by Using a Choice Box



In real-life applications, the choice boxes are used to build multiple-choice lists.

Setting the Behavior for a Choice Box

The application shown in Figure 7–3 provides a multiple-choice box with five options. When a particular language is selected, the corresponding greeting is rendered.

Figure 7–3 Multiple-Choice List

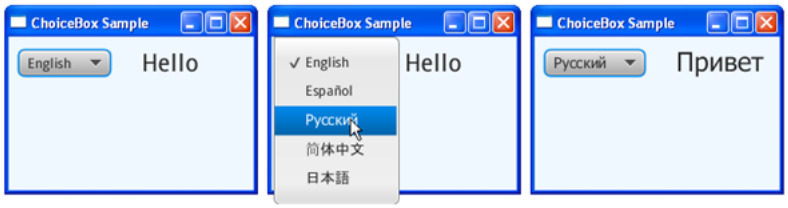


Figure 7–4 provides a code fragment to illustrate how an item selected from the choice box defines which greeting should be rendered.

Figure 7–4 Selecting a Choice Box Item

```
final String[] greetings = new String[]{"Hello", "Hola", "Привет", "你好",
    "こんにちは"};
final ChoiceBox cb = new ChoiceBox(FXCollections.observableArrayList(
    "English", "Español", "Русский", "简体中文", "日本語")
);

cb.getSelectionModel().selectedIndexProperty().addListener(new
     ChangeListener<Number>() {
         public void changed(ObservableValue ov,
             Number value, Number new_value) {
                 label.setText(greetings[new_value.intValue()]);
     }
});
```

A `ChangeListener<Number>` object detects the index of the currently selected item by consecutive calls of the `getSelectionModel` and `selectedIndexProperty` methods. The `getSelectionModel` method returns the selected item, and the `selectedIndexProperty` method returns the `SELECTED_INDEX` property of the cb choice box. As a result, the integer value as an index defines an element of the greetings array and specifies a `String` text value for the label. If, for example, a user selects the second item, which corresponds to Spanish, the `SELECTED_INDEX` is equal to 1 and "Hola" is selected from the greetings array. Thus, the label renders "Hola."

You can make the `ChoiceBox` control more informative by assigning a tooltip to it. A tooltip is a UI control that is available in the `javafx.scene.control` package. A tooltip can be applied to any of the JavaFX UI controls.

Applying a Tooltip

The `Tooltip` class provides a prefabricated tooltip that can be easily applied to a choice box (or any other control) by calling the `setTooltip` method shown in Example 7–3.

Example 7–3 Adding a Tooltip to a Choice Box

```
cb.setTooltip(new Tooltip("Select the language"));
```

Typically a user defines the text of the tooltip within a constructor of the `Tooltip` class. However, if the logic of your application requires UI to set the text dynamically, you can apply a tooltip by using an empty constructor and then assign the text to it by using the `setText` method.

After the tooltip is applied to the cb choice box, a user who positions the cursor over the choice box sees the image shown in Figure 7–5.

Figure 7–5 Choice Box with the Applied Tooltip

To further enhance your application, you can style the choice box with the CSS properties or apply visual effects or transformations.

Related API Documentation

- `ChoiceBox`
- `Tooltip`

8

Text Field

This chapter discusses the capabilities of the text field control.

The `TextField` class implements a UI control that accepts and displays text input. It provides capabilities to receive text input from a user. Along with another text input control, `PasswordField`, this class extends the `TextInput` class, a super class for all the text controls available through the JavaFX API.

Figure 8–1 shows a typical text field with a label.

Figure 8–1 Label and Text Field

Creating a Text Field

In Example 8–1, a text field is used in combination with a label to indicate the type of content that should be typed in the field.

Example 8–1 Creating a Text Field

```
Label label1 = new Label("Name:");
TextField textField = new TextField ();
HBox hb = new HBox();
hb.getChildren().addAll(label1, textField);
hb.setSpacing(10);
```

You can create an empty text field as shown in Example 8–1 or a text field with a particular text data in it. To create a text field with the predefined text, use the following constructor of the `TextField` class: `TextField("Hello World!")`. You can obtain the value of a text field at any time by calling the `getText` method.

You can apply the `setPrefColumnCount` method of the `TextInput` class to set the size of the text field, defined as the maximum number of characters it can display at one time.

Building the UI with Text Fields

Typically, the `TextField` objects are used in forms to create several text fields. The application in Figure 8–2 displays three text fields and processes the data that a user enters in them.

Figure 8–2 TextFieldSample Application

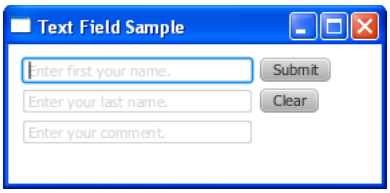
The code fragment in Example 8–2 creates the three text fields and two buttons, and adds them to the application's scene by using the `GridPane` container. This container is particularly handy when you need to implement a flexible layout for your UI controls.

Example 8–2 Adding Text Fields to the Application

```
//Creating a GridPane container
GridPane grid = new GridPane();
grid.setPadding(new Insets(10, 10, 10, 10));
grid.setVgap(5);
grid.setHgap(5);
//Defining the Name text field
final TextField name = new TextField();
name.setPromptText("Enter your first name.");
name.setPrefColumnCount(10);
name.getText();
GridPane.setConstraints(name, 0, 0);
grid.getChildren().add(name);
//Defining the Last Name text field
final TextField lastName = new TextField();
lastName.setPromptText("Enter your last name.");
GridPane.setConstraints(lastName, 0, 1);
grid.getChildren().add(lastName);
//Defining the Comment text field
final TextField comment = new TextField();
comment.setPrefColumnCount(15);
comment.setPromptText("Enter your comment.");
GridPane.setConstraints(comment, 0, 2);
grid.getChildren().add(comment);
//Defining the Submit button
Button submit = new Button("Submit");
GridPane.setConstraints(submit, 1, 0);
grid.getChildren().add(submit);
//Defining the Clear button
Button clear = new Button("Clear");
GridPane.setConstraints(clear, 1, 1);
grid.getChildren().add(clear);
```

Take a moment to study the code fragment. The `name`, `lastName`, and `comment` text fields are created by using empty constructors of the `TextField` class. Unlike Example 8–1, labels do not accompany the text fields in this code fragment. Instead, prompt captions notify users what type of data to enter in the text fields. The `setPromptText` method defines the string that appears in the text field when the application is started. When Example 8–2 is added to the application, it produces the output shown in Figure 8–3.

Figure 8–3 Three Text Fields with the Prompt Messages



The difference between the prompt text and the text entered in the text field is that the prompt text cannot be obtained through the `getText` method.

In real-life applications, data entered into the text fields is processed according to an application's logic as required by a specific business task. The next section explains how to use text fields to evaluate the entered data and generate a response to a user.

Processing Text Field Data

As mentioned earlier, the text data entered by a user into the text fields can be obtained by the `getText` method of the `TextInput` class.

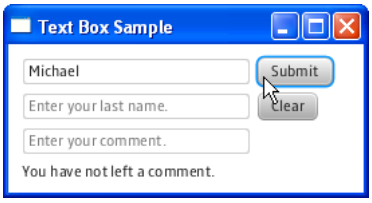
Study Example 8–3 to learn how to process the data of the `TextField` object.

Example 8–3 Defining Actions for the Submit and Clear Buttons

```
//Adding a Label
final Label label = new Label();
GridPane.setConstraints(label, 0, 3);
GridPane.setColumnSpan(label, 2);
grid.getChildren().add(label);

//Setting an action for the Submit button
submit.setOnAction(new EventHandler<ActionEvent>() {

@Override
    public void handle(ActionEvent e) {
        if ((comment.getText() != null && !comment.getText().isEmpty())) {
            label.setText(name.getText() + " " + lastName.getText() + ", "
                + "thank you for your comment!");
        } else {
            label.setText("You have not left a comment.");
        }
     }
 });

//Setting an action for the Clear button
clear.setOnAction(new EventHandler<ActionEvent>() {

@Override
    public void handle(ActionEvent e) {
        name.clear();
        lastName.clear();
        comment.clear();
        label.setText(null);
    }
});
```

The `Label` control added to the `GridPane` container renders an application's response to users. When a user clicks the Submit button, the `setOnAction` method checks the `comment` text field. If it contains a nonempty string, a thank-you message is rendered. Otherwise, the application notifies a user that the comment message has not been left yet, as shown in Figure 8–4.

Figure 8–4 The Comment Text Field Left Blank



When a user clicks the Clear button, the content is erased in all three text fields.

Review some helpful methods that you can use with text fields.

- `copy()` – transfers the currently selected range in the text to the clipboard, leaving the current selection.
- `cut()` – transfers the currently selected range in the text to the clipboard, removing the current selection.
- `paste()` – transfers the contents in the clipboard into this text, replacing the current selection.

Related API Documentation

- `TextField`
- `TextInputControl`

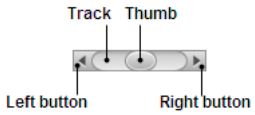
9

Scroll Bar

This chapter explains how to create scrollable panes by using the scroll bar control.

The `ScrollBar` class enables you to create scrollable panes and views in your application. Figure 9–1 shows the three areas of a scroll bar: the thumb, the right and left buttons (or down and up buttons), and the track.

Figure 9–1 Elements of the scroll bar



Creating a Scroll Bar

Take a moment to review the code fragment in Example 9–1.

Example 9–1 Simple Scroll Bar

```
ScrollBar sc = new ScrollBar();
sc.setMin(0);
sc.setMax(100);
sc.setValue(50);
```

The `setMin` and `setMax` methods define the minimum and maximum values represented by the scroll bar. When a user moves the thumb, the value of the scroll bar changes. In Example 9–1, the value equals 50, so when the application starts, the thumb is in the center of the scroll bar. By default, the scroll bar is oriented horizontally. However, you can set the vertical orientation by using the `setOrientation` method.

The user can click the left or right button (down or up button for the vertical orientation) to scroll by a unit increment. The `UNIT_INCREMENT` property specifies the amount by which the scroll bar is adjusted when a button is clicked. Another option is clicking within the track by a block increment. The `BLOCK_INCREMENT` property defines the amount by which the scroll bar is adjusted when the track of the bar is clicked.

In your application, you can use one of several scroll bars to scroll through graphical content that exceeds the borders of the available space.

Using a Scroll Bar in Your Application

Examine the scroll bar in action. The application shown in Example 9–2 implements a scrollable scene to view the images. The task of this application is to enable users to view the content of the vertical box, which is longer than the scene's height.

Example 9–2 Scrolling Through Multiple Images

```
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.geometry.Orientation;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.ScrollBar;
import javafx.scene.effect.DropShadow;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class Main extends Application {

    final ScrollBar sc = new ScrollBar();
    final Image[] images = new Image[5];
    final ImageView[] pics = new ImageView[5];
    final VBox vb = new VBox();
    DropShadow shadow = new DropShadow();

    @Override
    public void start(Stage stage) {
        Group root = new Group();
        Scene scene = new Scene(root, 180, 180);
        scene.setFill(Color.BLACK);
        stage.setScene(scene);
        stage.setTitle("Scrollbar");
        root.getChildren().addAll(vb, sc);

        shadow.setColor(Color.GREY);
        shadow.setOffsetX(2);
        shadow.setOffsetY(2);

        vb.setLayoutX(5);
        vb.setSpacing(10);

        sc.setLayoutX(scene.getWidth()-sc.getWidth());
        sc.setMin(0);
        sc.setOrientation(Orientation.VERTICAL);
        sc.setPrefHeight(180);
        sc.setMax(360);

        for (int i = 0; i < 5; i++) {
            final Image image = images[i] =
                new Image(getClass().getResourceAsStream(
                    "fw" +(i+1)+ ".jpg")
                );
            final ImageView pic = pics[i] =
                new ImageView(images[i]);
            pic.setEffect(shadow);
```

```
            vb.getChildren().add(pics[i]);
        }

        sc.valueProperty().addListener(new ChangeListener<Number>() {
            public void changed(ObservableValue<? extends Number> ov,
                Number old_val, Number new_val) {
                    vb.setLayoutY(-new_val.doubleValue());
            }
        });

        stage.show();
    }

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

The first lines of the code add a vertical box with the images and a scroll bar to the scene.

The `Y` coordinate of the vertical box changes when the `VALUE` property of the scroll bar is changed, so that each time the thumb is moved, or either a button or the track is clicked, the vertical box moves, as shown in Example 9–3.

Example 9–3 Implementing the Scrolling of the Vertical Box

```
sc.valueProperty().addListener(new ChangeListener<Number>() {
    public void changed(ObservableValue<? extends Number> ov,
        Number old_val, Number new_val) {
            vb.setLayoutY(-new_val.doubleValue());
        }
});
```

Compiling and running this application produces the output shown in Figure 9–2.

Figure 9–2 Scroll Bar Sample

This application shows one typical use of the `ScrollBar` class. You can also customize this class to create a scroll area within a scene. As for every UI control and every node, the scroll bar can be styled to change its appearance from the default implementation.

Related API Documentation

- `ScrollBar`
- JavaFX CSS Specification

10

Scroll Pane

In this chapter, you learn how to build scroll panes in your JavaFX applications.

Scroll panes provide a scrollable view of UI elements. This control enables the user to scroll the content by panning the viewport or by using scroll bars. A scroll pane with the default settings and the added image is shown in Figure 10–1.

Figure 10–1 Scroll Pane

Creating a Scroll Pane

Example 10–1 shows how to create this scroll pane in your application.

Example 10–1 Using a Scroll Pane to View an Image

```
Image roses = new Image(getClass().getResourceAsStream("roses.jpg"));
ScrollPane sp = new ScrollPane();
sp.setContent(new ImageView(roses));
```

The `setContent` method defines the node that is used as the content of this scroll pane. You can specify only one node. To create a scroll view with more than one component, use layout containers or the `Group` class. You can also specify the `true` value for the `setPannable` method to preview the image by clicking it and moving the mouse cursor. The position of the scroll bars changes accordingly.

Setting the Scroll Bar Policy for a Scroll Pane

The `ScrollPane` class provides a policy to determine when to display scroll bars: always, never, or only when they are needed. Use the `setHbarPolicy` and `setVbarPolicy` methods to specify the scroll bar policy for the horizontal and vertical scroll bars respectively. Thus, in Example 10–2 the vertical scroll bar will appear, but not the horizontal scroll bar.

Example 10–2 Setting the Horizontal and Vertical Scroll Bar Policies

```
sp.setHbarPolicy(ScrollBarPolicy.NEVER);
sp.setVbarPolicy(ScrollBarPolicy.ALWAYS);
```

As a result, you can only scroll the image vertically, as shown in Figure 10–2.

Figure 10–2 Disabling the Horizontal Scroll Bar

Resizing Components in the Scroll Pane

When designing a UI interface, you might need to resize the components, so that they match the width or height of the scroll pane. Set either the `setFitToWidth` or `setFitToHeight` method to `true` to match a particular dimension.

The scroll pane shown in Figure 10–3 contains radio buttons, a text box, and a password box. The size of the content exceeds the predefined size of the scroll pane and a vertical scroll bar appears. However, because the `setFitToWidth` method sets `true` for the scroll pane, the content shrinks in width and never scrolls horizontally.

Figure 10–3 Fitting the Width of the Scroll Pane

By default, both `FIT_TO_WIDTH` and `FIT_TO_HEIGHT` properties are `false`, and the resizable content keeps its original size. If you remove the `setFitToWidth` methods from the code of this application, you will see the output shown in Figure 10–4.

Figure 10–4 Default Properties for Fitting the Content

The `ScrollPane` class enables you to retrieve and set the current, minimum, and maximum values of the contents in the horizontal and vertical directions. Learn how to use them in your applications.

Sample Application with a Scroll Pane

Example 10–3 uses a scroll pane to display a vertical box with images. The `VVALUE` property of the `ScrollPane` class helps to identify the currently displayed image and to render the name of the image file.

Example 10–3 Using a Scroll Pane to View Images

```
package scrollpanesample;

import javafx.application.Application;
import javafx.beans.value.ChangeListener;
```

```
import javafx.beans.value.ObservableValue;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.ScrollPane;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.Priority;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class Main extends Application {

    final ScrollPane sp = new ScrollPane();
    final Image[] images = new Image[5];
    final ImageView[] pics = new ImageView[5];
    final VBox vb = new VBox();
    final Label fileName = new Label();
    final String [] imageNames = new String [] {"fw1.jpg", "fw2.jpg",
        "fw3.jpg", "fw4.jpg", "fw5.jpg"};

    @Override
    public void start(Stage stage) {
        VBox box = new VBox();
        Scene scene = new Scene(box, 180, 180);
        stage.setScene(scene);
        stage.setTitle("Scroll Pane");
        box.getChildren().addAll(sp, fileName);
        VBox.setVgrow(sp, Priority.ALWAYS);

        fileName.setLayoutX(30);
        fileName.setLayoutY(160);

        for (int i = 0; i < 5; i++) {
            images[i] = new Image(getClass().getResourceAsStream(imageNames[i]));
            pics[i] = new ImageView(images[i]);
            pics[i].setFitWidth(100);
            pics[i].setPreserveRatio(true);
            vb.getChildren().add(pics[i]);
        }

        sp.setVmax(440);
        sp.setPrefSize(115, 150);
        sp.setContent(vb);
        sp.vvalueProperty().addListener(new ChangeListener<Number>() {
            public void changed(ObservableValue<? extends Number> ov,
                Number old_val, Number new_val) {
                    fileName.setText(imageNames[(new_val.intValue() - 1)/100]);
            }
        });
        stage.show();
    }

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

Compiling and running this application produces the window shown in Figure 10–5.

Figure 10–5 Scrolling Images

The maximum value of the vertical scroll bar is equal to the height of the vertical box. The code fragment shown in Example 10–4 renders the name of the currently displayed image file.

Example 10–4 Tracking the Change of the Scroll Pane's Vertical Value

```
sp.vvalueProperty().addListener(new ChangeListener<Number>() {
    public void changed(ObservableValue<? extends Number> ov,
        Number old_val, Number new_val) {
            fileName.setText(imageNames[(new_val.intValue() - 1)/100]);
        }
});
```

The `ImageView` object limits the image height to 100 pixels. Therefore, when `new_val.intValue() - 1` is divided by 100, the result gives the index of the current image in the `imageNames` array.

In your application, you can also vary minimum and maximum values for vertical and horizontal scroll bars, thus dynamically updating your user interface.

Related API Documentation

- `ScrollPane`
- `ScrollBar`

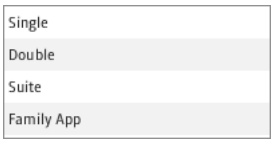
11

List View

In this chapter, you learn how to create lists in your JavaFX applications.

The `ListView` class represents a scrollable list of items. Figure 11–1 shows the list of available accommodation types in a hotel reservation system.

Figure 11–1 Simple List View



You can populate the list by defining its items with the `setItems` method. You can also create a view for the items in the list by applying the `setCellFactory` method.

Creating a List View

The code fragment in Example 11–1 implements the list with the `String` items shown in Figure 11–1.

Example 11–1 Creating a List View Control

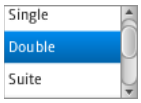
```
ListView<String> list = new ListView<String>();
ObservableList<String> items =FXCollections.observableArrayList (
    "Single", "Double", "Suite", "Family App");
list.setItems(items);
```

To alter the size and height of the list view control, use the `setPrefHeight` and `setPrefWidth` methods. Example 11–2 constrains the vertical list to 100 pixels wide by 70 pixels high, which results in the list shown in Figure 11–2.

Example 11–2 Setting Height and Width for a List View

```
list.setPrefWidth(100);
list.setPrefHeight(70);
```

Figure 11–2 Resized Vertical List



You can orient a `ListView` object horizontally by setting the orientation property to `Orientation.HORIZONTAL`. This can be done as follows: `list.setOrientation(Orientation.HORIZONTAL)`. The horizontal list with the same items as in Figure 11–1 is shown in Figure 11–3.

Figure 11–3 Horizontal List View Control

At any time, you can track the selection and focus of the `ListView` object with the `SelectionModel` and `FocusModel` classes. To obtain the current state of each item, use a combination of the following methods:

- `getSelectionModel().getSelectedIndex()` – Returns the index of the currently selected items in a single-selection mode
- `getSelectionModel().getSelectedItem()` – Returns the currently selected item
- `getFocusModel().getFocusedIndex()` – Returns the index of the currently focused item
- `getFocusModel().getFocusedItem()` – Returns the currently focused item

The default `SelectionModel` used when instantiating a `ListView` is an implementation of the `MultipleSelectionModel` abstract class. However, the default value of the `selectionMode` property is `SelectionMode.SINGLE`. To enable multiple selection in a default `ListView` instance, use the following sequence of calls:

```
listView.getSelectionModel().setSelectionMode(SelectionMode.MULT
IPLE);
```

Also note that `MultipleSelectionModel` has the `selectedItems` and `selectedIndices` properties, which are both observable lists that can be monitored to detect any multiple selections.

Populating a List View with Data

Example 11–1 shows the simplest way to populate a list view. To enhance your list, you can add data of various types by using the specific extensions of the `ListCell` class, such as `CheckBoxListCell`, `ChoiceBoxListCell`, `ComboBoxListCell`, and `TextFieldListCell`. These classes bring additional functionality to the basic list cell. Implementing cell factories for such classes enables developers to change data directly in the list view.

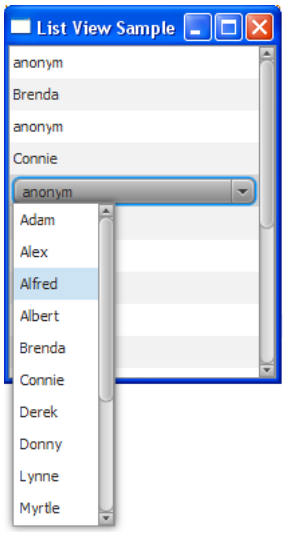
For example, the content of a list cell is not editable by default. However, the `ComboBoxListCell` class draws a combo box inside the list cell. This modification enables users to build a list of names by selecting them from a combo box, as shown in Example 11–3.

Example 11–3 Adding ComboBoxListCell Items to a List View

```
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.control.cell.ComboBoxListCell;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class ListViewSample extends Application {

    public static final ObservableList names =
        FXCollections.observableArrayList();
    public static final ObservableList data =
        FXCollections.observableArrayList();

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

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("List View Sample");

        final ListView listView = new ListView(data);
        listView.setPrefSize(200, 250);
        listView.setEditable(true);

        names.addAll(
             "Adam", "Alex", "Alfred", "Albert",
             "Brenda", "Connie", "Derek", "Donny",
             "Lynne", "Myrtle", "Rose", "Rudolph",
             "Tony", "Trudy", "Williams", "Zach"
        );

        for (int i = 0; i < 18; i++) {
            data.add("anonym");
        }

        listView.setItems(data);
        listView.setCellFactory(ComboBoxListCell.forListView(names));

        StackPane root = new StackPane();
        root.getChildren().add(listView);
        primaryStage.setScene(new Scene(root, 200, 250));
        primaryStage.show();
    }
}
```

The bold line in the example, calls the `setCellFactory` method to redefine the implementation of the list cell. When you compile and run this example, it produces the application window shown in Figure 11–4.

Figure 11–4 List View with the Combo Box Cells



Not only the cell factory mechanism enables you to apply the alternative implementations of the list cells, it can help you to totally customize the appearance of the cells.

Customizing the Content of a List View

Study the following application to learn how to generate the list items by using the cell factory. The application shown in Example 11–4 creates a list of color patterns.

Example 11–4 Creating a Cell Factory

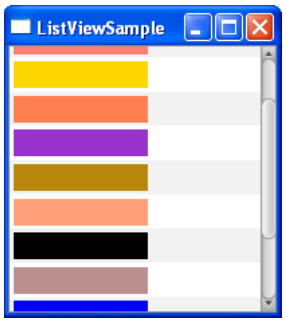
```
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.control.ListCell;
import javafx.scene.control.ListView;
import javafx.scene.layout.Priority;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;
import javafx.util.Callback;

public class ListViewSample extends Application {

    ListView<String> list = new ListView<String>();
    ObservableList<String> data = FXCollections.observableArrayList(
            "chocolate", "salmon", "gold", "coral", "darkorchid",
            "darkgoldenrod", "lightsalmon", "black", "rosybrown", "blue",
            "blueviolet", "brown");

    @Override
    public void start(Stage stage) {
        VBox box = new VBox();
        Scene scene = new Scene(box, 200, 200);
        stage.setScene(scene);
        stage.setTitle("ListViewSample");
```

```
        box.getChildren().addAll(list);
        VBox.setVgrow(list, Priority.ALWAYS);

        list.setItems(data);

        list.setCellFactory(new Callback<ListView<String>,
            ListCell<String>>() {
                @Override
                public ListCell<String> call(ListView<String> list) {
                    return new ColorRectCell();
                }
            }
        );

        stage.show();
    }

    static class ColorRectCell extends ListCell<String> {
        @Override
        public void updateItem(String item, boolean empty) {
            super.updateItem(item, empty);
            Rectangle rect = new Rectangle(100, 20);
            if (item != null) {
                rect.setFill(Color.web(item));
                setGraphic(rect);
            }
        }
    }

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

The cell factory produces `ListCell` objects. Every cell is associated with a single data item and renders a single "row" of the list view. The content that the cell represents through the `setGraphic` method can include other controls, text, shapes, or images. In this application, the list cell shows rectangles.

Compiling and running the application produces the window shown in Figure 11–5.

Figure 11–5 List of Color Patterns



You can scroll through the list, selecting and deselecting any of its items. You can also extend this application to fill the text label with the color pattern as shown in the next section.

Processing the List Item Selection

Modify the application code as shown in Example 11–5 to enable processing of the event when a particular list item is selected.

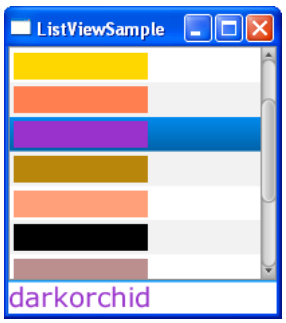
Example 11–5 Processing Events for a List Item

```
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.ListCell;
import javafx.scene.control.ListView;
import javafx.scene.layout.Priority;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Font;
import javafx.stage.Stage;
import javafx.util.Callback;

public class ListViewSample extends Application {

    ListView<String> list = new ListView<String>();
    ObservableList<String> data = FXCollections.observableArrayList(
            "chocolate", "salmon", "gold", "coral", "darkorchid",
            "darkgoldenrod", "lightsalmon", "black", "rosybrown", "blue",
            "blueviolet", "brown");
    final Label label = new Label();

    @Override
    public void start(Stage stage) {
        VBox box = new VBox();
        Scene scene = new Scene(box, 200, 200);
        stage.setScene(scene);
        stage.setTitle("ListViewSample");
        box.getChildren().addAll(list, label);
        VBox.setVgrow(list, Priority.ALWAYS);

        label.setLayoutX(10);
        label.setLayoutY(115);
        label.setFont(Font.font("Verdana", 20));

        list.setItems(data);

        list.setCellFactory(new Callback<ListView<String>,
            ListCell<String>>() {
                @Override
                public ListCell<String> call(ListView<String> list) {
                    return new ColorRectCell();
                }
            }
        );

        list.getSelectionModel().selectedItemProperty().addListener(
            new ChangeListener<String>() {
                public void changed(ObservableValue<? extends String> ov,
```

```
                    String old_val, String new_val) {
                        label.setText(new_val);
                        label.setTextFill(Color.web(new_val));
                }
            });
            stage.show();
        }

        static class ColorRectCell extends ListCell<String> {
            @Override
            public void updateItem(String item, boolean empty) {
                super.updateItem(item, empty);
                Rectangle rect = new Rectangle(100, 20);
                if (item != null) {
                    rect.setFill(Color.web(item));
                    setGraphic(rect);
                }
            }
        }

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

The `addListener` method called for the `selectedItemProperty` creates a new `ChangeListener<String>` object to handle changes of the selected item. If, for instance, the dark orchid item is selected, the label receives the "darkorchid" caption and is filled with the corresponding color. The output of the modified application is shown in Figure 11–6.

Figure 11–6 Selecting a Dark Orchid Color Pattern



Related API Documentation

- `ListView`
- `ListCell`
- `ComboBoxListCell`

12

Table View

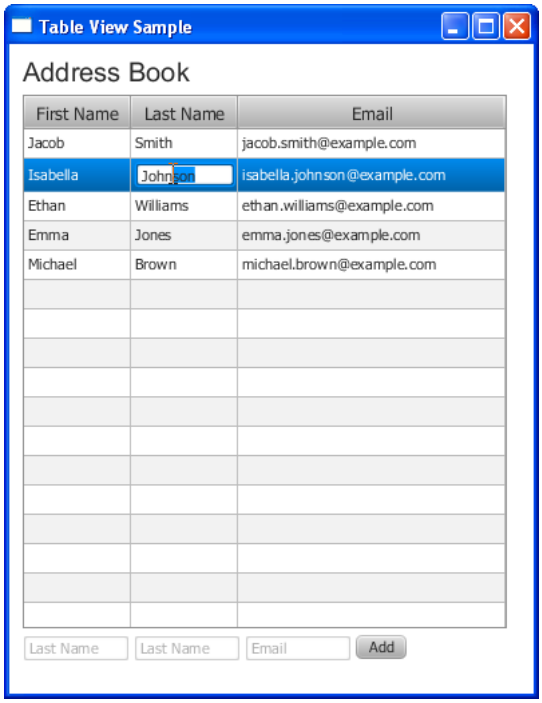
In this chapter, you learn how to perform basic operations with tables in JavaFX applications, such as adding a table, populating the table with data, and editing table rows.

Several classes in the JavaFX SDK API are designed to represent data in a tabular form. The most important classes for creating tables in JavaFX applications are `TableView`, `TableColumn`, and `TableCell`. You can populate a table by implementing the data model and by applying a cell factory.

The table classes provide built-in capabilities to sort data in columns and to resize columns when necessary.

Figure 12–1 shows a typical table representing contact information from an address book.

Figure 12–1 Table Sample



"

****

Creating a Table

The code fragment in Example 12–1 creates an empty table with three columns and adds it to the application scene.

Example 12–1 Adding a Table

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.stage.Stage;

public class TableViewSample extends Application {

    private TableView table = new TableView();
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage stage) {
        Scene scene = new Scene(new Group());
        stage.setTitle("Table View Sample");
        stage.setWidth(300);
        stage.setHeight(500);

        final Label label = new Label("Address Book");
        label.setFont(new Font("Arial", 20));

        table.setEditable(true);

        TableColumn firstNameCol = new TableColumn("First Name");
        TableColumn lastNameCol = new TableColumn("Last Name");
        TableColumn emailCol = new TableColumn("Email");

        table.getColumns().addAll(firstNameCol, lastNameCol, emailCol);

        final VBox vbox = new VBox();
        vbox.setSpacing(5);
        vbox.setPadding(new Insets(10, 0, 0, 10));
        vbox.getChildren().addAll(label, table);

        ((Group) scene.getRoot()).getChildren().addAll(vbox);

        stage.setScene(scene);
        stage.show();
    }
}
```

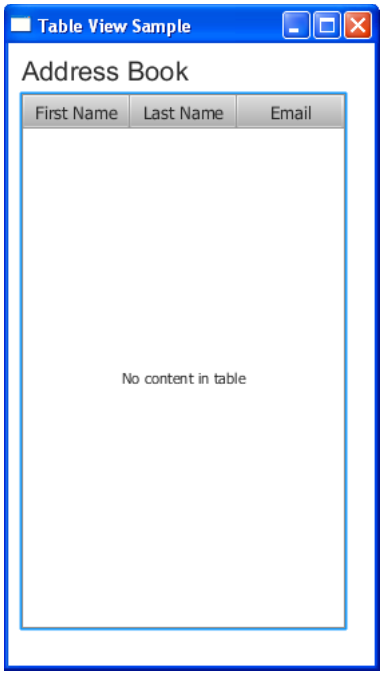
The table control is created by instantiating the `TableView` class. In Example 12–1, it is added to the `VBox` layout container, however, you can add it directly to the application scene.

Example 12–1 defines three columns to store the following information in an address book: a contact's first name and last name, and an email address. The columns are created by using the `TableColumn` class.

The `getColumns` method of the `TableView` class adds the previously created columns to the table. In your applications, you can use this method to dynamically add and remove columns.

Compiling and running this application produces the output shown in Figure 12–2.

Figure 12–2 Table Without Data



You can manage visibility of the columns by calling the `setVisible` method. For example, if the logic of your application requires hiding user email addresses, you can implement this task as follows: `emailCol.setVisible(false)`.

When the structure of your data requires a more complicated representation, you can create nested columns.

For example, suppose that the contacts in the address book have two email accounts. Then you need two columns to show the primary and the secondary email addresses. Create two subcolumns, and call the `getColumns` method on `emailCol` as shown in Example 12–2.

Example 12–2 Creating Nested Columns

```
TableColumn firstEmailCol = new TableColumn("Primary");
TableColumn secondEmailCol = new TableColumn("Secondary");

emailCol.getColumns().addAll(firstEmailCol, secondEmailCol);
```

After you have added these lines to Example 12–1, and compiled and run the application code, the table appears as shown in Figure 12–3.

Figure 12–3 Table with Nested Columns

Although the table is added to the application, the standard caption "No content in table" appears, because no data is defined. Instead of showing this caption, you can use the `setPlaceholder` method to specify a `Node` object to appear in an empty table.

Defining the Data Model

When you create a table in a JavaFX application, it is a best practice to implement a class that defines the data model and provides methods and fields to further work with the table. Example 12–3 creates the `Person` class to define data in an address book.

Example 12–3 Creating the Person Class

```
public static class Person {
    private final SimpleStringProperty firstName;
    private final SimpleStringProperty lastName;
    private final SimpleStringProperty email;

    private Person(String fName, String lName, String email) {
        this.firstName = new SimpleStringProperty(fName);
        this.lastName = new SimpleStringProperty(lName);
        this.email = new SimpleStringProperty(email);
    }

    public String getFirstName() {
        return firstName.get();
    }
    public void setFirstName(String fName) {
        firstName.set(fName);
```

```
    }

    public String getLastName() {
        return lastName.get();
    }
    public void setLastName(String fName) {
        lastName.set(fName);
    }

    public String getEmail() {
        return email.get();
    }
    public void setEmail(String fName) {
        email.set(fName);
    }

}
```

The `firstName`, `lastName`, and `email` string properties are created to enable the referencing of a particular data element.

Additionally, the `get` and `set` methods are provided for each data element. Thus, for example, the `getFirstName` method returns the value of the `firstName` property, and the `setFirstName` method specifies a value for this property.

When the data model is outlined in the `Person` class, you can create an `ObservableList` array and define as many data rows as you would like to show in your table. The code fragment in Example 12–4 implements this task.

Example 12–4 Defining Table Data in an Observable List

```
final ObservableList<Person> data = FXCollections.observableArrayList(
    new Person("Jacob", "Smith", "jacob.smith@example.com"),
    new Person("Isabella", "Johnson", "isabella.johnson@example.com"),
    new Person("Ethan", "Williams", "ethan.williams@example.com"),
    new Person("Emma", "Jones", "emma.jones@example.com"),
    new Person("Michael", "Brown", "michael.brown@example.com")
);
```

The next step is to associate the data with the table columns. You can do this through the properties defined for each data element, as shown in Example 12–5.

Example 12–5 Setting Data Properties to Columns

```
firstNameCol.setCellValueFactory(
    new PropertyValueFactory<Person,String>("firstName")
);
lastNameCol.setCellValueFactory(
    new PropertyValueFactory<Person,String>("lastName")
);
emailCol.setCellValueFactory(
    new PropertyValueFactory<Person,String>("email")
);
```

The `setCellValueFactory` method specifies a cell factory for each column. The cell factories are implemented by using the `PropertyValueFactory` class, which uses the `firstName`, `lastName`, and `email` properties of the table columns as references to the corresponding methods of the `Person` class.

When the data model is defined, and the data is added and associated with the columns, you can add the data to the table by using the `setItems` method of the `TableView` class: `table.setItems(data)`.

Because the `ObservableList` object enables the tracking of any changes to its elements, the `TableView` content automatically updates whenever the data changes.

Examine the application code shown in Example 12–6.

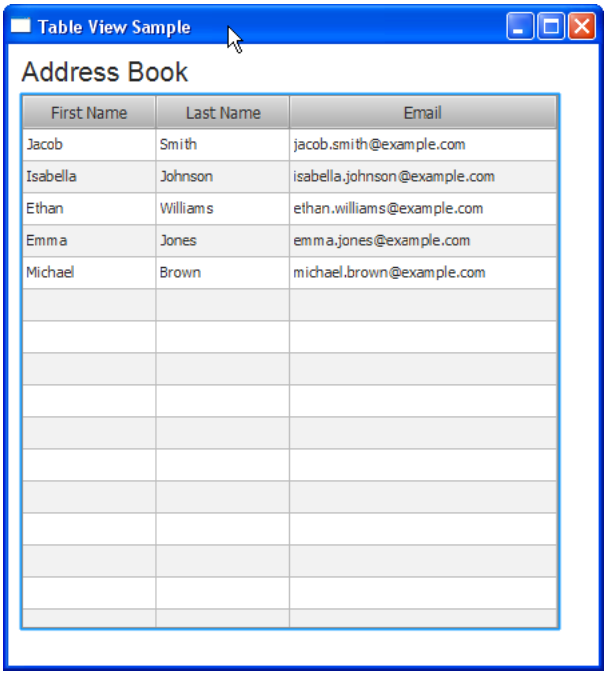
Example 12–6 Creating a Table and Adding Data to It

```
import javafx.application.Application;
import javafx.beans.property.SimpleStringProperty;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.geometry.Insets;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.TextField;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.stage.Stage;

public class TableViewSample extends Application {

    private TableView<Person> table = new TableView<Person>();
    private final ObservableList<Person> data =
        FXCollections.observableArrayList(
            new Person("Jacob", "Smith", "jacob.smith@example.com"),
            new Person("Isabella", "Johnson", "isabella.johnson@example.com"),
            new Person("Ethan", "Williams", "ethan.williams@example.com"),
            new Person("Emma", "Jones", "emma.jones@example.com"),
            new Person("Michael", "Brown", "michael.brown@example.com")
        );

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

    @Override
    public void start(Stage stage) {
        Scene scene = new Scene(new Group());
        stage.setTitle("Table View Sample");
        stage.setWidth(450);
        stage.setHeight(500);

        final Label label = new Label("Address Book");
        label.setFont(new Font("Arial", 20));

        table.setEditable(true);

        TableColumn firstNameCol = new TableColumn("First Name");
        firstNameCol.setMinWidth(100);
        firstNameCol.setCellValueFactory(
                new PropertyValueFactory<Person, String>("firstName"));

        TableColumn lastNameCol = new TableColumn("Last Name");
```

```
        lastNameCol.setMinWidth(100);
        lastNameCol.setCellValueFactory(
                new PropertyValueFactory<Person, String>("lastName"));

        TableColumn emailCol = new TableColumn("Email");
        emailCol.setMinWidth(200);
        emailCol.setCellValueFactory(
                new PropertyValueFactory<Person, String>("email"));

        table.setItems(data);
        table.getColumns().addAll(firstNameCol, lastNameCol, emailCol);

        final VBox vbox = new VBox();
        vbox.setSpacing(5);
        vbox.setPadding(new Insets(10, 0, 0, 10));
        vbox.getChildren().addAll(label, table);

        ((Group) scene.getRoot()).getChildren().addAll(vbox);

        stage.setScene(scene);
        stage.show();
    }

    public static class Person {

        private final SimpleStringProperty firstName;
        private final SimpleStringProperty lastName;
        private final SimpleStringProperty email;

        private Person(String fName, String lName, String email) {
            this.firstName = new SimpleStringProperty(fName);
            this.lastName = new SimpleStringProperty(lName);
            this.email = new SimpleStringProperty(email);
        }

        public String getFirstName() {
            return firstName.get();
        }

        public void setFirstName(String fName) {
            firstName.set(fName);
        }

        public String getLastName() {
            return lastName.get();
        }

        public void setLastName(String fName) {
            lastName.set(fName);
        }

        public String getEmail() {
            return email.get();
        }

        public void setEmail(String fName) {
            email.set(fName);
        }
    }
}
```

When you compile and run this application code, the table shown in Figure 12–4 appears.

Figure 12–4 Table Populated with Data



Adding New Rows

The table in Figure 12–4 contains five rows of data, which cannot be modified so far.

You can use text fields to enter new values into the First Name, Last Name, and Email columns. The Text Field control enables your application to receive text input from a user. Example 12–7 creates three text fields, defines the prompt text for each field, and creates the Add button.

Example 12–7 Using Text Fields to Enter New Items in the Table

```
final TextField addFirstName = new TextField();
addFirstName.setPromptText("First Name");
addFirstName.setMaxWidth(firstNameCol.getPrefWidth());
final TextField addLastName = new TextField();
addLastName.setMaxWidth(lastNameCol.getPrefWidth());
addLastName.setPromptText("Last Name");
final TextField addEmail = new TextField();
addEmail.setMaxWidth(emailCol.getPrefWidth());
addEmail.setPromptText("Email");

final Button addButton = new Button("Add");
addButton.setOnAction(new EventHandler<ActionEvent>() {
    @Override public void handle(ActionEvent e) {
        data.add(new Person(
            addFirstName.getText(),
            addLastName.getText(),
            addEmail.getText()
        ));
```

```
        addFirstName.clear();
        addLastName.clear();
        addEmail.clear();
    }
});
```

When a user clicks the Add button, the values entered in the text fields are included in a `Person` constructor and added to the `data` observable list. Thus, the new entry with contact information appears in the table.

Examine the application code shown in Example 12–8.

Example 12–8 Table with the Text Fields to Enter New Items

```
import javafx.application.Application;
import javafx.beans.property.SimpleStringProperty;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.TextField;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.stage.Stage;

public class FileChooserSample extends Application {

    private TableView<Person> table = new TableView<Person>();
    private final ObservableList<Person> data =
            FXCollections.observableArrayList(
            new Person("Jacob", "Smith", "jacob.smith@example.com"),
            new Person("Isabella", "Johnson", "isabella.johnson@example.com"),
            new Person("Ethan", "Williams", "ethan.williams@example.com"),
            new Person("Emma", "Jones", "emma.jones@example.com"),
            new Person("Michael", "Brown", "michael.brown@example.com"));
    final HBox hb = new HBox();

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

    @Override
    public void start(Stage stage) {
        Scene scene = new Scene(new Group());
        stage.setTitle("Table View Sample");
        stage.setWidth(450);
        stage.setHeight(550);

        final Label label = new Label("Address Book");
        label.setFont(new Font("Arial", 20));

        table.setEditable(true);
```

```
    TableColumn firstNameCol = new TableColumn("First Name");
    firstNameCol.setMinWidth(100);
    firstNameCol.setCellValueFactory(
            new PropertyValueFactory<Person, String>("firstName"));

    TableColumn lastNameCol = new TableColumn("Last Name");
    lastNameCol.setMinWidth(100);
    lastNameCol.setCellValueFactory(
            new PropertyValueFactory<Person, String>("lastName"));

    TableColumn emailCol = new TableColumn("Email");
    emailCol.setMinWidth(200);
    emailCol.setCellValueFactory(
            new PropertyValueFactory<Person, String>("email"));

    table.setItems(data);
    table.getColumns().addAll(firstNameCol, lastNameCol, emailCol);

    final TextField addFirstName = new TextField();
    addFirstName.setPromptText("First Name");
    addFirstName.setMaxWidth(firstNameCol.getPrefWidth());
    final TextField addLastName = new TextField();
    addLastName.setMaxWidth(lastNameCol.getPrefWidth());
    addLastName.setPromptText("Last Name");
    final TextField addEmail = new TextField();
    addEmail.setMaxWidth(emailCol.getPrefWidth());
    addEmail.setPromptText("Email");

    final Button addButton = new Button("Add");
    addButton.setOnAction(new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent e) {
            data.add(new Person(
                    addFirstName.getText(),
                    addLastName.getText(),
                    addEmail.getText()));
            addFirstName.clear();
            addLastName.clear();
            addEmail.clear();
        }
    });

    hb.getChildren().addAll(addFirstName, addLastName, addEmail, addButton);
    hb.setSpacing(3);

    final VBox vbox = new VBox();
    vbox.setSpacing(5);
    vbox.setPadding(new Insets(10, 0, 0, 10));
    vbox.getChildren().addAll(label, table, hb);

    ((Group) scene.getRoot()).getChildren().addAll(vbox);

    stage.setScene(scene);
    stage.show();
}

public static class Person {

    private final SimpleStringProperty firstName;
```

```
        private final SimpleStringProperty lastName;
        private final SimpleStringProperty email;

        private Person(String fName, String lName, String email) {
            this.firstName = new SimpleStringProperty(fName);
            this.lastName = new SimpleStringProperty(lName);
            this.email = new SimpleStringProperty(email);
        }

        public String getFirstName() {
            return firstName.get();
        }

        public void setFirstName(String fName) {
            firstName.set(fName);
        }

        public String getLastName() {
            return lastName.get();
        }

        public void setLastName(String fName) {
            lastName.set(fName);
        }

        public String getEmail() {
            return email.get();
        }

        public void setEmail(String fName) {
            email.set(fName);
        }
    }
}
```

This application does not provide any filters to check if, for example, an email address was entered in an incorrect format. You can provide such functionality when you develop your own application.

The current implementation also does not check to determine if the empty values are entered. If no values are provided, clicking the Add button inserts an empty row in the table.

Figure 12–5 demonstrates how a user adds a new row of data.

Figure 12–5 Adding Contact Information to the Address Book

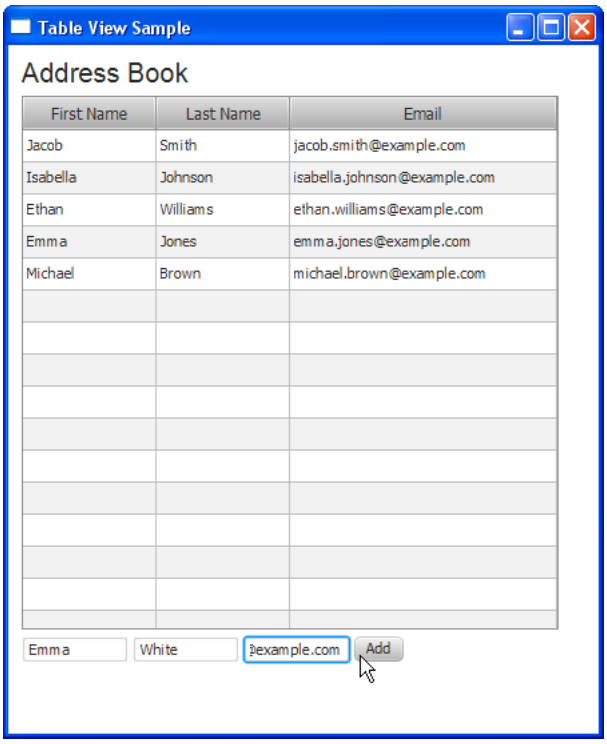
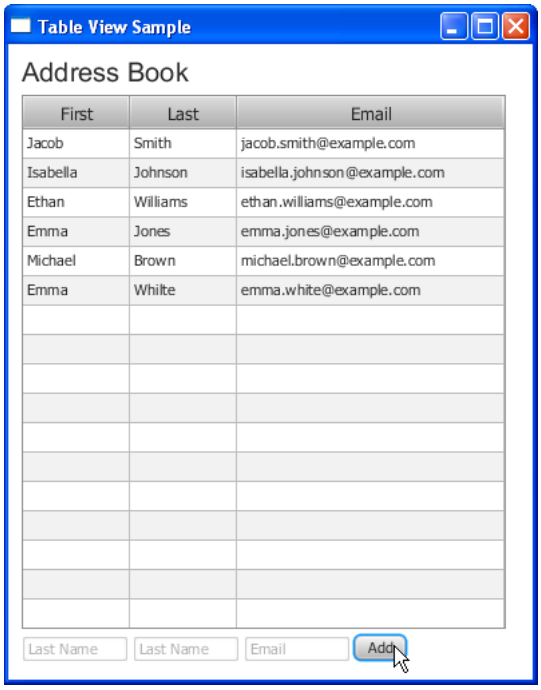


Figure 12–6 shows the table after the Add button is clicked. The contact details of Emma White now appear in the table.

Figure 12–6 Newly Added Entry



Sorting Data in Columns

The `TableView` class provides built-in capabilities to sort data in columns. Users can alter the order of data by clicking column headers. The first click enables the ascending sorting order, the second click enables descending sorting order, and the third click disables sorting. By default, no sorting is applied.

Users can sort multiple columns in a table and specify the priority of each column in the sort operation. To sort multiple columns, the user presses the Shift key while clicking the header of each column to be sorted.

In Figure 12–7, an ascending sort order is applied to the first names, while last names are sorted in a descending order. Note that the first column has priority over the second column.

Figure 12–7 Sorting Multiple Columns

As the application developer, you can set sorting preferences for each column in your application by applying the `setSortType` method. You can specify both ascending and descending type. For example, use the following code line to set the descending type of sorting for the emailCol column:
`emailCol.setSortType(TableColumn.SortType.DESCENDING);`.

You can also specify which columns to sort by adding and removing `TableColumn` instances from the `TableView.sortOrder` observable list. The order of columns in this list represents the sort priority (for example, the zero item has higher priority than the first item).

To prohibit sorting of data, call the `setSortable(false)` method on the column.

Editing Data in the Table

The `TableView` class not only renders tabular data, but it also provides capabilities to edit it. Use the `setEditable` method to enable editing of the table content.

Use the `setCellFactory` method to reimplement the table cell as a text field with the help of the `TextFieldTableCell` class. The `setOnEditCommit` method processes editing and assigns the updated value to the corresponding table cell.

Example 12–9 shows how to apply these methods to process cell editing in the First Name, Last Name, and Email columns.

Example 12–9 Implementing Cell Editing

```
firstNameCol.setCellFactory(TextFieldTableCell.forTableColumn());
firstNameCol.setOnEditCommit(
    new EventHandler<CellEditEvent<Person, String>>() {
        @Override
        public void handle(CellEditEvent<Person, String> t) {
            ((Person) t.getTableView().getItems().get(
                t.getTablePosition().getRow())
                ).setFirstName(t.getNewValue());
        }
    }
);

lastNameCol.setCellFactory(TextFieldTableCell.forTableColumn());
lastNameCol.setOnEditCommit(
    new EventHandler<CellEditEvent<Person, String>>() {
        @Override
        public void handle(CellEditEvent<Person, String> t) {
            ((Person) t.getTableView().getItems().get(
                t.getTablePosition().getRow())
                ).setLastName(t.getNewValue());
        }
    }
);

emailCol.setCellFactory(TextFieldTableCell.forTableColumn());
emailCol.setOnEditCommit(
    new EventHandler<CellEditEvent<Person, String>>() {
        @Override
        public void handle(CellEditEvent<Person, String> t) {
            ((Person) t.getTableView().getItems().get(
                t.getTablePosition().getRow())
                ).setEmail(t.getNewValue());
        }
    }
);
```

The complete code of the application shown in Example 12–10.

Example 12–10 TableViewSample with Enabled Cell Editing

```
import javafx.application.Application;
import javafx.beans.property.SimpleStringProperty;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableColumn.CellEditEvent;
import javafx.scene.control.TableView;
```

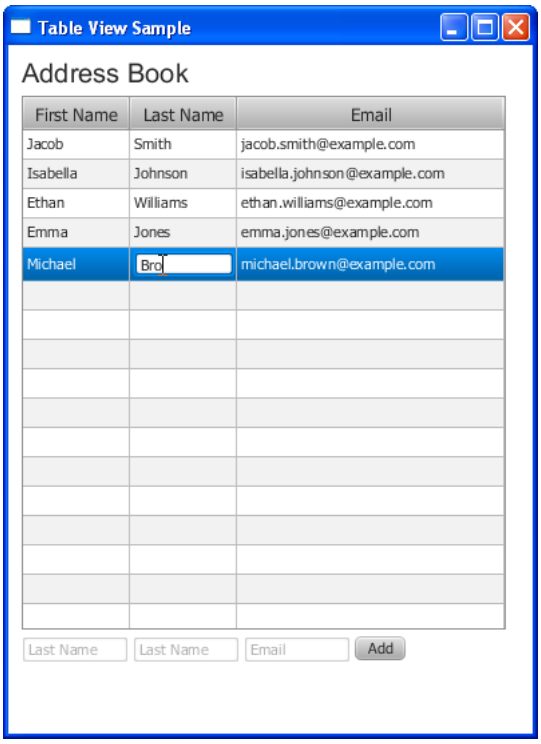
```
import javafx.scene.control.TextField;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.control.cell.TextFieldTableCell;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.stage.Stage;

public class TableViewSample extends Application {

    private TableView<Person> table = new TableView<Person>();
    private final ObservableList<Person> data =
            FXCollections.observableArrayList(
            new Person("Jacob", "Smith", "jacob.smith@example.com"),
            new Person("Isabella", "Johnson", "isabella.johnson@example.com"),
            new Person("Ethan", "Williams", "ethan.williams@example.com"),
            new Person("Emma", "Jones", "emma.jones@example.com"),
            new Person("Michael", "Brown", "michael.brown@example.com"));
    final HBox hb = new HBox();

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

    @Override
    public void start(Stage stage) {
        Scene scene = new Scene(new Group());
        stage.setTitle("Table View Sample");
        stage.setWidth(450);
        stage.setHeight(550);

        final Label label = new Label("Address Book");
        label.setFont(new Font("Arial", 20));

        table.setEditable(true);

        TableColumn firstNameCol = new TableColumn("First Name");
        firstNameCol.setMinWidth(100);
        firstNameCol.setCellValueFactory(
            new PropertyValueFactory<Person, String>("firstName"));
        firstNameCol.setCellFactory(TextFieldTableCell.forTableColumn());
        firstNameCol.setOnEditCommit(
            new EventHandler<CellEditEvent<Person, String>>() {
                @Override
                public void handle(CellEditEvent<Person, String> t) {
                    ((Person) t.getTableView().getItems().get(
                            t.getTablePosition().getRow())
                            ).setFirstName(t.getNewValue());
                }
            }
        );


        TableColumn lastNameCol = new TableColumn("Last Name");
        lastNameCol.setMinWidth(100);
        lastNameCol.setCellValueFactory(
            new PropertyValueFactory<Person, String>("lastName"));
        lastNameCol.setCellFactory(TextFieldTableCell.forTableColumn());
        lastNameCol.setOnEditCommit(
            new EventHandler<CellEditEvent<Person, String>>() {
```

```
        @Override
        public void handle(CellEditEvent<Person, String> t) {
            ((Person) t.getTableView().getItems().get(
                t.getTablePosition().getRow())
                ).setLastName(t.getNewValue());
        }
    }
);

TableColumn emailCol = new TableColumn("Email");
emailCol.setMinWidth(200);
emailCol.setCellValueFactory(
    new PropertyValueFactory<Person, String>("email"));
emailCol.setCellFactory(TextFieldTableCell.forTableColumn());
emailCol.setOnEditCommit(
    new EventHandler<CellEditEvent<Person, String>>() {
        @Override
        public void handle(CellEditEvent<Person, String> t) {
            ((Person) t.getTableView().getItems().get(
                t.getTablePosition().getRow())
                ).setEmail(t.getNewValue());
        }
    }
);

table.setItems(data);
table.getColumns().addAll(firstNameCol, lastNameCol, emailCol);

final TextField addFirstName = new TextField();
addFirstName.setPromptText("First Name");
addFirstName.setMaxWidth(firstNameCol.getPrefWidth());
final TextField addLastName = new TextField();
addLastName.setMaxWidth(lastNameCol.getPrefWidth());
addLastName.setPromptText("Last Name");
final TextField addEmail = new TextField();
addEmail.setMaxWidth(emailCol.getPrefWidth());
addEmail.setPromptText("Email");

final Button addButton = new Button("Add");
addButton.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent e) {
        data.add(new Person(
                addFirstName.getText(),
                addLastName.getText(),
                addEmail.getText()));
        addFirstName.clear();
        addLastName.clear();
        addEmail.clear();
    }
});

hb.getChildren().addAll(addFirstName, addLastName, addEmail, addButton);
hb.setSpacing(3);

final VBox vbox = new VBox();
vbox.setSpacing(5);
vbox.setPadding(new Insets(10, 0, 0, 10));
vbox.getChildren().addAll(label, table, hb);
```

```
        ((Group) scene.getRoot()).getChildren().addAll(vbox);

        stage.setScene(scene);
        stage.show();
    }

    public static class Person {

        private final SimpleStringProperty firstName;
        private final SimpleStringProperty lastName;
        private final SimpleStringProperty email;

        private Person(String fName, String lName, String email) {
            this.firstName = new SimpleStringProperty(fName);
            this.lastName = new SimpleStringProperty(lName);
            this.email = new SimpleStringProperty(email);
        }

        public String getFirstName() {
            return firstName.get();
        }

        public void setFirstName(String fName) {
            firstName.set(fName);
        }

        public String getLastName() {
            return lastName.get();
        }

        public void setLastName(String fName) {
            lastName.set(fName);
        }

        public String getEmail() {
            return email.get();
        }

        public void setEmail(String fName) {
            email.set(fName);
        }
    }
}
```

In Figure 12–8, the user is editing the last name of Michael Brown. To edit a table cell, the user enters the new value in the cell, and then presses the Enter key. The cell is not modified until the Enter key is pressed. This behavior is determined by the implementation of the `TextField` class.

Figure 12–8 Editing a Table Cell



"

****

Note that the default implementation of the `TextField` control requires that users press the Enter key to commit the edit. You can redefine the `TextField` behavior to commit the edit on the focus change, which is an expected user experience. Try the modified code in to implement such an alternative behavior.

Example 12–11 Alternative Solution Of Cell Editing

```
import javafx.application.Application;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TableCell;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableColumn.CellEditEvent;
import javafx.scene.control.TableView;
import javafx.scene.control.TextField;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
```

```
import javafx.scene.text.Font;
import javafx.stage.Stage;
import javafx.util.Callback;

public class TableViewSample extends Application {

    private TableView<Person> table = new TableView<Person>();
    private final ObservableList<Person> data =
            FXCollections.observableArrayList(
            new Person("Jacob", "Smith", "jacob.smith@example.com"),
            new Person("Isabella", "Johnson", "isabella.johnson@example.com"),
            new Person("Ethan", "Williams", "ethan.williams@example.com"),
            new Person("Emma", "Jones", "emma.jones@example.com"),
            new Person("Michael", "Brown", "michael.brown@example.com"));
    final HBox hb = new HBox();

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

    @Override
    public void start(Stage stage) {
        Scene scene = new Scene(new Group());
        stage.setTitle("Table View Sample");
        stage.setWidth(450);
        stage.setHeight(550);

        final Label label = new Label("Address Book");
        label.setFont(new Font("Arial", 20));

        table.setEditable(true);
        Callback<TableColumn, TableCell> cellFactory =
             new Callback<TableColumn, TableCell>() {
                 public TableCell call(TableColumn p) {
                    return new EditingCell();
                 }
             };

        TableColumn firstNameCol = new TableColumn("First Name");
        firstNameCol.setMinWidth(100);
        firstNameCol.setCellValueFactory(
            new PropertyValueFactory<Person, String>("firstName"));
        firstNameCol.setCellFactory(cellFactory);
        firstNameCol.setOnEditCommit(
            new EventHandler<CellEditEvent<Person, String>>() {
                @Override
                public void handle(CellEditEvent<Person, String> t) {
                    ((Person) t.getTableView().getItems().get(
                        t.getTablePosition().getRow())
                        ).setFirstName(t.getNewValue());
                }
             }
        );


        TableColumn lastNameCol = new TableColumn("Last Name");
        lastNameCol.setMinWidth(100);
        lastNameCol.setCellValueFactory(
            new PropertyValueFactory<Person, String>("lastName"));
        lastNameCol.setCellFactory(cellFactory);
```

```
lastNameCol.setOnEditCommit(
    new EventHandler<CellEditEvent<Person, String>>() {
        @Override
        public void handle(CellEditEvent<Person, String> t) {
            ((Person) t.getTableView().getItems().get(
                t.getTablePosition().getRow())
                ).setLastName(t.getNewValue());
        }
    }
);

TableColumn emailCol = new TableColumn("Email");
emailCol.setMinWidth(200);
emailCol.setCellValueFactory(
    new PropertyValueFactory<Person, String>("email"));
emailCol.setCellFactory(cellFactory);
emailCol.setOnEditCommit(
    new EventHandler<CellEditEvent<Person, String>>() {
        @Override
        public void handle(CellEditEvent<Person, String> t) {
            ((Person) t.getTableView().getItems().get(
                t.getTablePosition().getRow())
                ).setEmail(t.getNewValue());
        }
    }
);

table.setItems(data);
table.getColumns().addAll(firstNameCol, lastNameCol, emailCol);

final TextField addFirstName = new TextField();
addFirstName.setPromptText("First Name");
addFirstName.setMaxWidth(firstNameCol.getPrefWidth());
final TextField addLastName = new TextField();
addLastName.setMaxWidth(lastNameCol.getPrefWidth());
addLastName.setPromptText("Last Name");
final TextField addEmail = new TextField();
addEmail.setMaxWidth(emailCol.getPrefWidth());
addEmail.setPromptText("Email");

final Button addButton = new Button("Add");
addButton.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent e) {
        data.add(new Person(
                addFirstName.getText(),
                addLastName.getText(),
                addEmail.getText()));
        addFirstName.clear();
        addLastName.clear();
        addEmail.clear();
    }
});

hb.getChildren().addAll(addFirstName, addLastName, addEmail, addButton);
hb.setSpacing(3);

final VBox vbox = new VBox();
vbox.setSpacing(5);
vbox.setPadding(new Insets(10, 0, 0, 10));
```

```
        vbox.getChildren().addAll(label, table, hb);

        ((Group) scene.getRoot()).getChildren().addAll(vbox);

        stage.setScene(scene);
        stage.show();
    }

    public static class Person {

        private final SimpleStringProperty firstName;
        private final SimpleStringProperty lastName;
        private final SimpleStringProperty email;

        private Person(String fName, String lName, String email) {
            this.firstName = new SimpleStringProperty(fName);
            this.lastName = new SimpleStringProperty(lName);
            this.email = new SimpleStringProperty(email);
        }

        public String getFirstName() {
            return firstName.get();
        }

        public void setFirstName(String fName) {
            firstName.set(fName);
        }

        public String getLastName() {
            return lastName.get();
        }

        public void setLastName(String fName) {
            lastName.set(fName);
        }

        public String getEmail() {
            return email.get();
        }

        public void setEmail(String fName) {
            email.set(fName);
        }
    }

    class EditingCell extends TableCell<Person, String> {

        private TextField textField;

        public EditingCell() {
        }

        @Override
        public void startEdit() {
            if (!isEmpty()) {
                super.startEdit();
                createTextField();
                setText(null);
                setGraphic(textField);
                textField.selectAll();
```

```
            }
        }

        @Override
        public void cancelEdit() {
            super.cancelEdit();

            setText((String) getItem());
            setGraphic(null);
        }

        @Override
        public void updateItem(String item, boolean empty) {
            super.updateItem(item, empty);

            if (empty) {
                setText(null);
                setGraphic(null);
            } else {
                if (isEditing()) {
                    if (textField != null) {
                        textField.setText(getString());
                    }
                    setText(null);
                    setGraphic(textField);
                } else {
                    setText(getString());
                    setGraphic(null);
                }
            }
        }

        private void createTextField() {
            textField = new TextField(getString());
            textField.setMinWidth(this.getWidth() - this.getGraphicTextGap()* 2);
            textField.focusedProperty().addListener(new ChangeListener<Boolean>(){
                @Override
                public void changed(ObservableValue<? extends Boolean> arg0,
                    Boolean arg1, Boolean arg2) {
                        if (!arg2) {
                            commitEdit(textField.getText());
                        }
                }
            });
        }

        private String getString() {
            return getItem() == null ? "" : getItem().toString();
        }
    }
}
```

Note that this approach might become redundant in future releases as the `TextFieldTableCell` implementation is being evolved to provide better user experience.

Adding Maps of Data to the Table

Starting JavaFX SDK 2.2, you can add the `Map` data to the table. Use the `MapValueFactory` class as shown in Example 12–12 to display the map of student IDs in the table.

Example 12–12 Adding Map Data to the Table

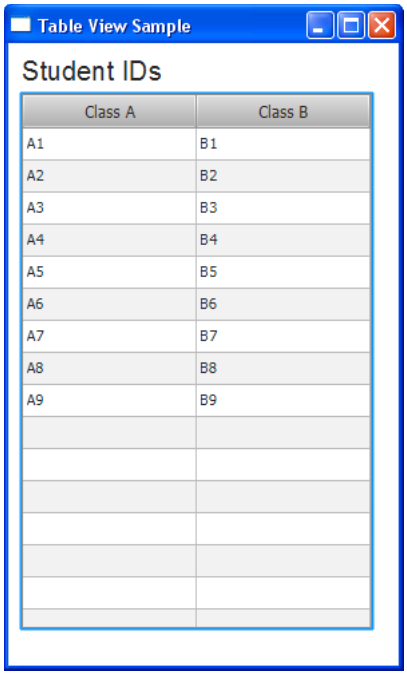
```
import java.util.HashMap;
import java.util.Map;
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.geometry.Insets;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TableCell;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.cell.MapValueFactory;
import javafx.scene.control.cell.TextFieldTableCell;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.stage.Stage;
import javafx.util.Callback;
import javafx.util.StringConverter;

public class TableViewSample extends Application {

    public static final String Column1MapKey = "A";
    public static final String Column2MapKey = "B";

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

    @Override
    public void start(Stage stage) {
        Scene scene = new Scene(new Group());
        stage.setTitle("Table View Sample");
        stage.setWidth(300);
        stage.setHeight(500);

        final Label label = new Label("Student IDs");
        label.setFont(new Font("Arial", 20));

        TableColumn<Map, String> firstDataColumn = new TableColumn<>("Class A");
        TableColumn<Map, String> secondDataColumn = new TableColumn<>("Class B");

        firstDataColumn.setCellValueFactory(new MapValueFactory(Column1MapKey));
        firstDataColumn.setMinWidth(130);
        secondDataColumn.setCellValueFactory(new MapValueFactory(Column2MapKey));
        secondDataColumn.setMinWidth(130);

        TableView table_view = new TableView<>(generateDataInMap());

        table_view.setEditable(true);
        table_view.getSelectionModel().setCellSelectionEnabled(true);
        table_view.getColumns().setAll(firstDataColumn, secondDataColumn);
```

```
            Callback<TableColumn<Map, String>, TableCell<Map, String>>
                cellFactoryForMap = new Callback<TableColumn<Map, String>,
                    TableCell<Map, String>>() {
                        @Override
                        public TableCell call(TableColumn p) {
                            return new TextFieldTableCell(new StringConverter() {
                                @Override
                                public String toString(Object t) {
                                    return t.toString();
                                }
                                @Override
                                public Object fromString(String string) {
                                    return string;
                                }
                            });
                        }
            };
            firstDataColumn.setCellFactory(cellFactoryForMap);
            secondDataColumn.setCellFactory(cellFactoryForMap);

            final VBox vbox = new VBox();

            vbox.setSpacing(5);
            vbox.setPadding(new Insets(10, 0, 0, 10));
            vbox.getChildren().addAll(label, table_view);

            ((Group) scene.getRoot()).getChildren().addAll(vbox);

            stage.setScene(scene);

            stage.show();
        }

        private ObservableList<Map> generateDataInMap() {
            int max = 10;
            ObservableList<Map> allData = FXCollections.observableArrayList();
            for (int i = 1; i < max; i++) {
                Map<String, String> dataRow = new HashMap<>();

                String value1 = "A" + i;
                String value2 = "B" + i;

                dataRow.put(Column1MapKey, value1);
                dataRow.put(Column2MapKey, value2);

                allData.add(dataRow);
            }
            return allData;
        }
    }
```

The `MapValueFactory` class implements the `Callback` interface, and it is designed specifically to be used within cell factories of table columns. In Example 12–12, the dataRow hash map presents a single row in the `TableView` object. The map has two `String` keys: Column1MapKey and Column2MapKey to map the corresponding values in the first and second columns. The `setCellValueFactory` method called for the table columns populates them with data matching with a particular key, so that the first column contains values that correspond to the "A" key and second column contains the values that correspond to the "B" key.

When you compile and run this application, it produces the output shown in Figure 12–9.

Figure 12–9 TableVIew with Map Data



Related API Documentation

- `TableView`
- `TableColumn`
- `TableCell`
- `TextField`
- `TextFieldTableCell`
- `MapValueFactory`
- `Button`

13

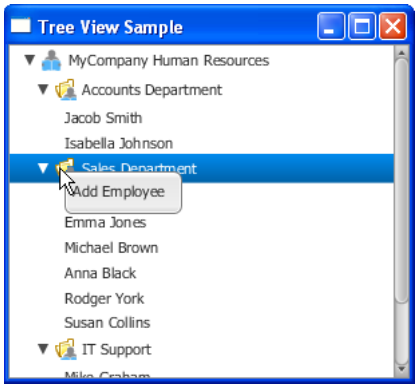
Tree View

In this chapter you can learn how to build tree structures in your JavaFX application, add items to the tree views, process events, and customize the tree cells by implementing and applying cell factories.

The `TreeView` class of the `javafx.scene.control` package provides a view of hierarchical structures. In each tree the highest object in the hierarchy is called the "root." The root contains several child items, which can have children as well. An item without children is called "leaf."

Figure 13–1 shows a screen capture of an application with a tree view.

Figure 13–1 Tree View Sample



Creating Tree Views

When you build a tree structure in your JavaFX applications, you typically need to instantiate the `TreeView` class, define several `TreeItem` objects, make one of the tree items the root, add the root to the tree view and other tree items to the root.

You can accompany each tree item with a graphical icon by using the corresponding constructor of the `TreeItem` class or by calling the `setGraphic` method. The recommended size for icons is 16x16, but in fact, any `Node` object can be set as the icon and it will be fully interactive.

Example 13–1 is an implementation of a simple tree view with the root and five leaves.

Example 13–1 Creating a Tree View

```
import javafx.application.Application;
import javafx.scene.Node;
```

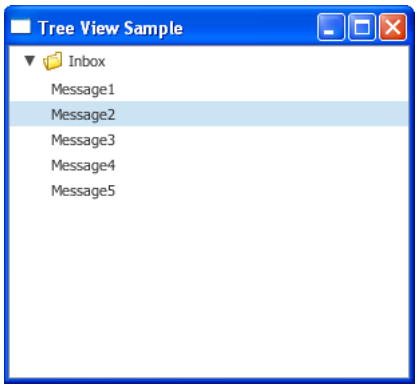
```
import javafx.scene.Scene;
import javafx.scene.control.TreeItem;
import javafx.scene.control.TreeView;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class TreeViewSample extends Application {

    private final Node rootIcon = new ImageView(
        new Image(getClass().getResourceAsStream("folder_16.png"))
    );

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

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Tree View Sample");

        TreeItem<String> rootItem = new TreeItem<String> ("Inbox", rootIcon);
        rootItem.setExpanded(true);
        for (int i = 1; i < 6; i++) {
           TreeItem<String> item = new TreeItem<String> ("Message" + i);
            rootItem.getChildren().add(item);
        }
        TreeView<String> tree = new TreeView<String> (rootItem);
        StackPane root = new StackPane();
        root.getChildren().add(tree);
        primaryStage.setScene(new Scene(root, 300, 250));
        primaryStage.show();
    }
}
```

All the tree items created within the `for` loop are added to the root item by calling the `getChildren` and `add` methods. You can also use the `addAll` method instead of the `add` method to include all the previously created tree items at once.

You can specify the root of the tree within the constructor of the `TreeView` class when you create a new `TreeView` object as shown in Example 13–1, or you can set it by calling the `setRoot` method of the `TreeView` class.

The `setExpanded` method called on the root item defines the initial appearance of the tree view item. By default, all `TreeItem` instances are collapsed, and must be manually expanded if required. Pass the `true` value to the `setExpanded` method, so that the root tree item looks expanded when the application starts, as shown in Figure 13–2.

Figure 13–2 Tree View with Five Tree Items



Example 13–1 creates a simple tree view with the `String` items. However, a tree structure can contain items of different types. Use the following generic notation of the `TreeItem` constructor to define application-specific data represented by a tree item: `TreeItem<T> (T value)`. The `T` value can specify any object, such as UI controls or custom components.

Unlike the `TreeView` class, the `TreeItem` class does not extend the `Node` class. Therefore, you cannot apply any visual effects or add menus to the tree items. Use the cell factory mechanism to overcome this obstacle and define as much custom behavior for the tree items as your application requires.

Implementing Cell Factories

The cell factory mechanism is used for generating `TreeCell` instances to represent a single `TreeItem` in the `TreeView`. Using cell factories is particularly helpful when your application operates with an excessive amount of data that is changed dynamically or added on demand.

Consider an application that visualizes human resources data of a given company, and enables users to modify employee details and add new employees.

Example 13–2 creates the `Employee` class and arranges employees in groups according to their departments.

Example 13–2 Creating a Model of the Human Resources Tree View

```
import java.util.Arrays;
import java.util.List;
import javafx.application.Application;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.control.TreeItem;
import javafx.scene.control.TreeView;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

import javafx.beans.property.SimpleStringProperty;
import javafx.scene.layout.VBox;

public class TreeViewSample extends Application {
```

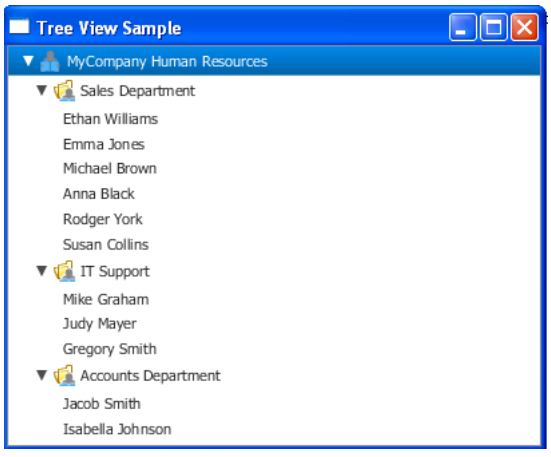
```
private final Node rootIcon =
    new ImageView(new Image(getClass().getResourceAsStream("root.png")));
private final Image depIcon =
    new Image(getClass().getResourceAsStream("department.png"));
List<Employee> employees = Arrays.<Employee>asList(
        new Employee("Ethan Williams", "Sales Department"),
        new Employee("Emma Jones", "Sales Department"),
        new Employee("Michael Brown", "Sales Department"),
        new Employee("Anna Black", "Sales Department"),
        new Employee("Rodger York", "Sales Department"),
        new Employee("Susan Collins", "Sales Department"),
        new Employee("Mike Graham", "IT Support"),
        new Employee("Judy Mayer", "IT Support"),
        new Employee("Gregory Smith", "IT Support"),
        new Employee("Jacob Smith", "Accounts Department"),
        new Employee("Isabella Johnson", "Accounts Department"));
TreeItem<String> rootNode =
    new TreeItem<String>("MyCompany Human Resources", rootIcon);

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

@Override
public void start(Stage stage) {
    rootNode.setExpanded(true);
    for (Employee employee : employees) {
        TreeItem<String> empLeaf = new TreeItem<String>(employee.getName());
        boolean found = false;
        for (TreeItem<String> depNode : rootNode.getChildren()) {
            if (depNode.getValue().contentEquals(employee.getDepartment())){
                depNode.getChildren().add(empLeaf);
                found = true;
                break;
            }
        }
        if (!found) {
            TreeItem<String> depNode = new TreeItem<String>(
                employee.getDepartment(),
                new ImageView(depIcon)
            );
            rootNode.getChildren().add(depNode);
            depNode.getChildren().add(empLeaf);
        }
    }

    stage.setTitle("Tree View Sample");
    VBox box = new VBox();
    final Scene scene = new Scene(box, 400, 300);
    scene.setFill(Color.LIGHTGRAY);

    TreeView<String> treeView = new TreeView<String>(rootNode);

    box.getChildren().add(treeView);
    stage.setScene(scene);
    stage.show();
}

public static class Employee {
```

```
        private final SimpleStringProperty name;
        private final SimpleStringProperty department;

        private Employee(String name, String department) {
            this.name = new SimpleStringProperty(name);
            this.department = new SimpleStringProperty(department);
        }

        public String getName() {
            return name.get();
        }

        public void setName(String fName) {
            name.set(fName);
        }

        public String getDepartment() {
            return department.get();
        }

        public void setDepartment(String fName) {
            department.set(fName);
        }
    }
}
```

Each `Employee` object in Example 13–2 has two properties: `name` and `department`. `TreeItem` objects corresponding to the employees are referred as tree leaves, whereas the tree items corresponding to the departments are referred to as tree items with children. The name of the new department to be created is retrieved from an `Employee` object by calling the `getDepartment` method.

When you compile and run this application, it creates the window shown in Figure 13–3.

Figure 13–3 List of Employees in the Tree View Sample Application



With Example 13–2, you can preview the tree view and its items, but you cannot change the existing items or add any new items. Example 13–3 shows a modified version of the application with the cell factory implemented. The modified application enables you to change the name of an employee.

Example 13–3 Implementing a Cell Factory

```
import java.util.Arrays;
import java.util.List;
import javafx.application.Application;
import javafx.event.EventHandler;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.control.TextField;
import javafx.scene.control.TreeCell;
import javafx.scene.control.TreeItem;
import javafx.scene.control.TreeView;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.input.KeyCode;
import javafx.scene.input.KeyEvent;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.util.Callback;

import javafx.beans.property.SimpleStringProperty;
import javafx.scene.layout.VBox;

public class TreeViewSample extends Application {

    private final Node rootIcon =
        new ImageView(new Image(getClass().getResourceAsStream("root.png")));
    private final Image depIcon =
        new Image(getClass().getResourceAsStream("department.png"));
    List<Employee> employees = Arrays.<Employee>asList(
            new Employee("Ethan Williams", "Sales Department"),
            new Employee("Emma Jones", "Sales Department"),
            new Employee("Michael Brown", "Sales Department"),
            new Employee("Anna Black", "Sales Department"),
            new Employee("Rodger York", "Sales Department"),
            new Employee("Susan Collins", "Sales Department"),
            new Employee("Mike Graham", "IT Support"),
            new Employee("Judy Mayer", "IT Support"),
            new Employee("Gregory Smith", "IT Support"),
            new Employee("Jacob Smith", "Accounts Department"),
            new Employee("Isabella Johnson", "Accounts Department"));
    TreeItem<String> rootNode =
        new TreeItem<String>("MyCompany Human Resources", rootIcon);

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

    @Override
    public void start(Stage stage) {
        rootNode.setExpanded(true);
        for (Employee employee : employees) {
            TreeItem<String> empLeaf = new TreeItem<String>(employee.getName());
            boolean found = false;
            for (TreeItem<String> depNode : rootNode.getChildren()) {
                if (depNode.getValue().contentEquals(employee.getDepartment())){
                    depNode.getChildren().add(empLeaf);
                    found = true;
                    break;
                }
            }
```

```
            if (!found) {
                TreeItem<String> depNode = new TreeItem<String>(
                    employee.getDepartment(),
                    new ImageView(depIcon)
                );
                rootNode.getChildren().add(depNode);
                depNode.getChildren().add(empLeaf);
            }
        }

        stage.setTitle("Tree View Sample");
        VBox box = new VBox();
        final Scene scene = new Scene(box, 400, 300);
        scene.setFill(Color.LIGHTGRAY);

        TreeView<String> treeView = new TreeView<String>(rootNode);
        treeView.setEditable(true);
        treeView.setCellFactory(new Callback<TreeView<String>,TreeCell<String>>(){
            @Override
            public TreeCell<String> call(TreeView<String> p) {
                return new TextFieldTreeCellImpl();
            }
        });

        box.getChildren().add(treeView);
        stage.setScene(scene);
        stage.show();
    }

    private final class TextFieldTreeCellImpl extends TreeCell<String> {

        private TextField textField;

        public TextFieldTreeCellImpl() {
        }

        @Override
        public void startEdit() {
            super.startEdit();

            if (textField == null) {
                createTextField();
            }
            setText(null);
            setGraphic(textField);
            textField.selectAll();
        }

        @Override
        public void cancelEdit() {
            super.cancelEdit();
            setText((String) getItem());
            setGraphic(getTreeItem().getGraphic());
        }

        @Override
        public void updateItem(String item, boolean empty) {
            super.updateItem(item, empty);

            if (empty) {
```

```
                setText(null);
                setGraphic(null);
            } else {
                if (isEditing()) {
                    if (textField != null) {
                        textField.setText(getString());
                    }
                    setText(null);
                    setGraphic(textField);
                } else {
                    setText(getString());
                    setGraphic(getTreeItem().getGraphic());
                }
            }
        }

        private void createTextField() {
            textField = new TextField(getString());
            textField.setOnKeyReleased(new EventHandler<KeyEvent>() {

                @Override
                public void handle(KeyEvent t) {
                    if (t.getCode() == KeyCode.ENTER) {
                        commitEdit(textField.getText());
                    } else if (t.getCode() == KeyCode.ESCAPE) {
                        cancelEdit();
                    }
                }
            });
        }

        private String getString() {
            return getItem() == null ? "" : getItem().toString();
        }
    }

    public static class Employee {

        private final SimpleStringProperty name;
        private final SimpleStringProperty department;

        private Employee(String name, String department) {
            this.name = new SimpleStringProperty(name);
            this.department = new SimpleStringProperty(department);
        }

        public String getName() {
            return name.get();
        }

        public void setName(String fName) {
            name.set(fName);
        }

        public String getDepartment() {
            return department.get();
        }

        public void setDepartment(String fName) {
            department.set(fName);
```

```
        }
    }
}
```

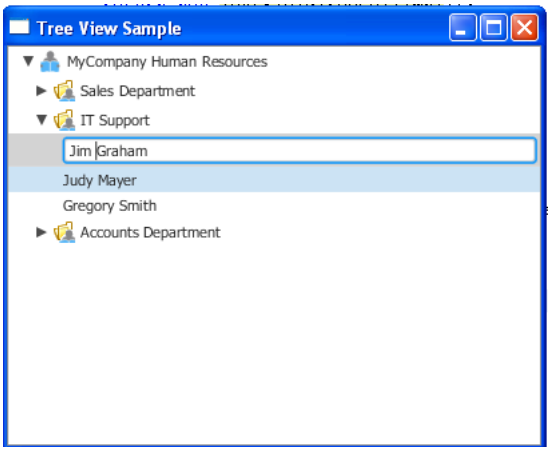
The `setCellFactory` method called on the `treeView` object overrides the `TreeCell` implementation and redefines the tree items as specified in the `TextFieldTreeCellImpl` class.

The `TextFieldTreeCellImpl` class creates a `TextField` object for each tree item and provides the methods to process editing events.

Note that you must explicitly call the `setEditable(true)` method on the tree view to enable editing all its items.

Compile and run the application in Example 13–3. Then try to click the employees in the tree and change their names. Figure 13–4 captures the moment of editing a tree item in the IT Support department.

Figure 13–4 Changing an Employee Name



Adding New Tree Items on Demand

Modify the Tree View Sample application so that a human resources representative can add new employees. Use the bold code lines of Example 13–4 for your reference. These lines add a context menu to the tree items that correspond to the departments. When the Add Employee menu item is selected, the new tree item is added as a leaf to the current department.

Use the `isLeaf` method to distinguish between department tree items and employee tree items.

Example 13–4 Adding New Tree Items

```
import java.util.Arrays;
import java.util.List;
import javafx.application.Application;
import javafx.event.Event;
import javafx.event.EventHandler;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.control.TextField;
import javafx.scene.control.TreeCell;
import javafx.scene.control.TreeItem;
```

```
import javafx.scene.control.TreeView;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.input.KeyCode;
import javafx.scene.input.KeyEvent;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.util.Callback;

import javafx.beans.property.SimpleStringProperty;
import javafx.scene.control.ContextMenu;
import javafx.scene.control.MenuItem;
import javafx.scene.layout.VBox;

public class TreeViewSample extends Application {

    private final Node rootIcon =
        new ImageView(new Image(getClass().getResourceAsStream("root.png")));
    private final Image depIcon =
        new Image(getClass().getResourceAsStream("department.png"));
    List<Employee> employees = Arrays.<Employee>asList(
            new Employee("Ethan Williams", "Sales Department"),
            new Employee("Emma Jones", "Sales Department"),
            new Employee("Michael Brown", "Sales Department"),
            new Employee("Anna Black", "Sales Department"),
            new Employee("Rodger York", "Sales Department"),
            new Employee("Susan Collins", "Sales Department"),
            new Employee("Mike Graham", "IT Support"),
            new Employee("Judy Mayer", "IT Support"),
            new Employee("Gregory Smith", "IT Support"),
            new Employee("Jacob Smith", "Accounts Department"),
            new Employee("Isabella Johnson", "Accounts Department"));
    TreeItem<String> rootNode =
        new TreeItem<String>("MyCompany Human Resources", rootIcon);

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

    @Override
    public void start(Stage stage) {
        rootNode.setExpanded(true);
        for (Employee employee : employees) {
            TreeItem<String> empLeaf = new TreeItem<String>(employee.getName());
            boolean found = false;
            for (TreeItem<String> depNode : rootNode.getChildren()) {
                if (depNode.getValue().contentEquals(employee.getDepartment())){
                    depNode.getChildren().add(empLeaf);
                    found = true;
                    break;
                }
            }
            if (!found) {
                TreeItem depNode = new TreeItem(employee.getDepartment(),
                    new ImageView(depIcon)
                );
                rootNode.getChildren().add(depNode);
                depNode.getChildren().add(empLeaf);
            }
        }
```

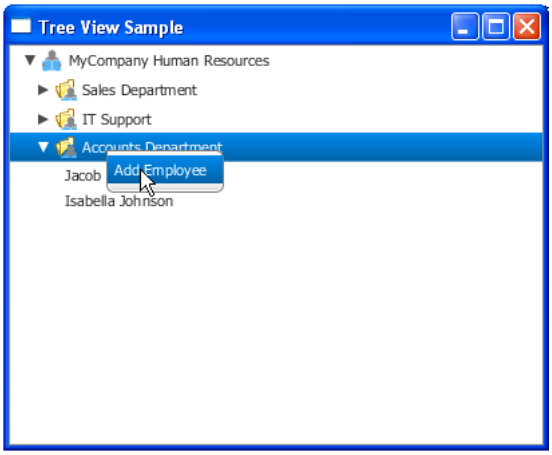
```
        stage.setTitle("Tree View Sample");
        VBox box = new VBox();
        final Scene scene = new Scene(box, 400, 300);
        scene.setFill(Color.LIGHTGRAY);

        TreeView<String> treeView = new TreeView<String>(rootNode);
        treeView.setEditable(true);
        treeView.setCellFactory(new Callback<TreeView<String>,TreeCell<String>>(){
            @Override
            public TreeCell<String> call(TreeView<String> p) {
                return new TextFieldTreeCellImpl();
            }
        });

        box.getChildren().add(treeView);
        stage.setScene(scene);
        stage.show();
    }

    private final class TextFieldTreeCellImpl extends TreeCell<String> {

        private TextField textField;
        private ContextMenu addMenu = new ContextMenu();

        public TextFieldTreeCellImpl() {
            MenuItem addMenuItem = new MenuItem("Add Employee");
            addMenu.getItems().add(addMenuItem);
            addMenuItem.setOnAction(new EventHandler() {
                public void handle(Event t) {
                    TreeItem newEmployee =
                        new TreeItem<String>("New Employee");
                            getTreeItem().getChildren().add(newEmployee);
                }
            });
        }

        @Override
        public void startEdit() {
            super.startEdit();

            if (textField == null) {
                createTextField();
            }
            setText(null);
            setGraphic(textField);
            textField.selectAll();
        }

        @Override
        public void cancelEdit() {
            super.cancelEdit();

            setText((String) getItem());
            setGraphic(getTreeItem().getGraphic());
        }

        @Override
        public void updateItem(String item, boolean empty) {
            super.updateItem(item, empty);
```

```
            if (empty) {
                setText(null);
                setGraphic(null);
            } else {
                if (isEditing()) {
                    if (textField != null) {
                        textField.setText(getString());
                    }
                    setText(null);
                    setGraphic(textField);
                } else {
                    setText(getString());
                    setGraphic(getTreeItem().getGraphic());
                    if (
                        !getTreeItem().isLeaf()&&getTreeItem().getParent()!= null
                    ){
                        setContextMenu(addMenu);
                    }
                }
            }
        }

        private void createTextField() {
            textField = new TextField(getString());
            textField.setOnKeyReleased(new EventHandler<KeyEvent>() {

                @Override
                public void handle(KeyEvent t) {
                    if (t.getCode() == KeyCode.ENTER) {
                        commitEdit(textField.getText());
                    } else if (t.getCode() == KeyCode.ESCAPE) {
                        cancelEdit();
                    }
                }
            });

        }

        private String getString() {
            return getItem() == null ? "" : getItem().toString();
        }
    }

    public static class Employee {

        private final SimpleStringProperty name;
        private final SimpleStringProperty department;

        private Employee(String name, String department) {
            this.name = new SimpleStringProperty(name);
            this.department = new SimpleStringProperty(department);
        }

        public String getName() {
            return name.get();
        }

        public void setName(String fName) {
            name.set(fName);
```

```
        }

        public String getDepartment() {
            return department.get();
        }

        public void setDepartment(String fName) {
            department.set(fName);
        }
    }
}
```

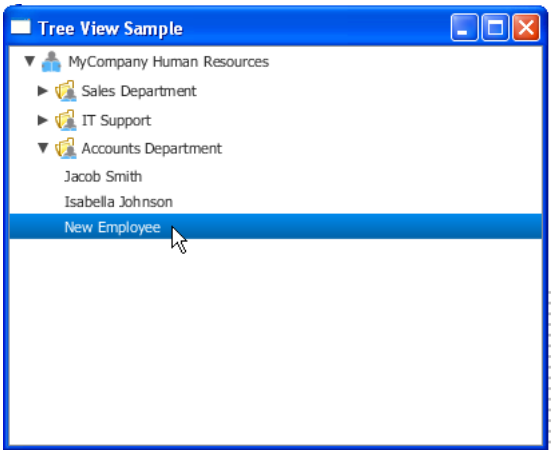
Compile and run the application. Then select a department in the tree structure and right-click it. The context menu appears, as shown in Figure 13–5.

Figure 13–5 Context Menu for Adding New Employees



When you select the Add Employee menu item from the context menu, the new record is added to the current department. Figure 13–6 shows a new tree item added to the Accounts Department.

Figure 13–6 Newly Added Employee



Because editing is enabled for the tree items, you can change the default "New Employee" value to the appropriate name.

Using Tree Cell Editors

Starting JavaFX SDK 2.2, you can use the following tree cell editors available in the API: `CheckBoxTreeCell`, `ChoiceBoxTreeCell`, `ComboBoxTreeCell`, `TextFieldTreeCell`. There classes extend the `TreeCell` implementation to render a particular control inside the cell.

Example 13–5 demonstrates the use of the `CheckBoxTreeCell` class in the UI that builds a hierarchical structure of checkboxes.

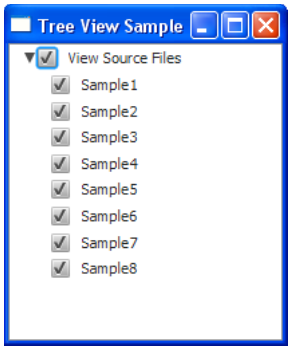
Example 13–5 Using the CheckBoxTreeCell Class

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.control.cell.CheckBoxTreeCell;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class TreeViewSample extends Application {

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

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Tree View Sample");

        CheckBoxTreeItem<String> rootItem =
            new CheckBoxTreeItem<String>("View Source Files");
        rootItem.setExpanded(true);

        final TreeView tree = new TreeView(rootItem);
        tree.setEditable(true);

        tree.setCellFactory(CheckBoxTreeCell.<String>forTreeView());
        for (int i = 0; i < 8; i++) {
            final CheckBoxTreeItem<String> checkBoxTreeItem =
                new CheckBoxTreeItem<String>("Sample" + (i+1));
                    rootItem.getChildren().add(checkBoxTreeItem);
        }

        tree.setRoot(rootItem);
        tree.setShowRoot(true);

        StackPane root = new StackPane();
        root.getChildren().add(tree);
        primaryStage.setScene(new Scene(root, 300, 250));
        primaryStage.show();
    }
}
```

Example 13–5 builds the tree view by using the `CheckBoxTreeItem` class instead of the TreeItem. The `CheckBoxTreeItem` class was specifically designed to support the selected, unselected, and indeterminate states in tree structures. A `CheckBoxTreeItem` instance can be independent or dependent. If a `CheckBoxTreeItem` instance is independent, any changes to its selection state do not impact its parent and child `CheckBoxTreeItem` instances. By default, all the `ChechBoxTreeItem` instances are dependent.

Compile and run Example 13–5, then select the View Source Files item. You should see the output shown in Figure 13–7, where all the child items are selected.

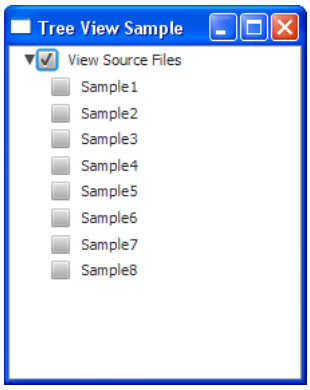
Figure 13–7 Dependent CheckBoxTreeItem



To make a `CheckBoxTreeItem` instance independent, use the `setIndependent` method: `rootItem.setIndependent(true);`.

When you run the TreeViewSample application, its behavior should change as shown in Figure 13–8.

Figure 13–8 Independent CheckBoxTreeItem



Related API Documentation

- `TreeView`
- `TreeItem`
- `TreeCell`
- `Cell`
- `TextField`
- `CheckBoxTreeCell`
- `CheckBoxTreeItem`

14

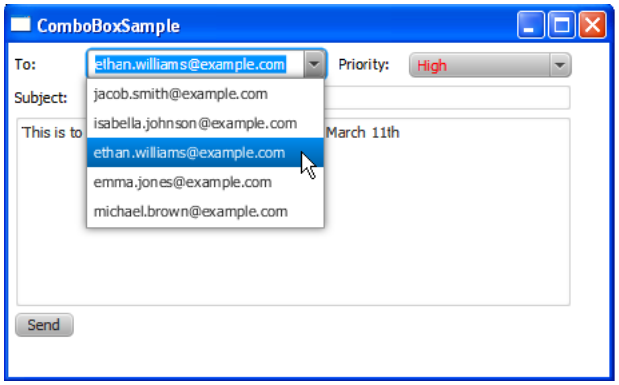
Combo Box

This chapter explains how to use combo boxes in your JavaFX application. It discusses editable and uneditable combo boxes, teaches you how to track changes in the editable combo boxes and handle events on them, and explains how to use cell factories to alter the default implementation of a combo box.

A combo box is a typical element of a user interface that enables users to choose one of several options. A combo box is helpful when the number of items to show exceeds some limit, because it can add scrolling to the drop down list, unlike a choice box. If the number of items does not exceed a certain limit, developers can decide whether a combo box or a choice box better suits their needs.

You can create a combo box in the JavaFX application by using the `ComboBox` class of the JavaFX API. Figure 14–1 shows an application with two combo boxes.

Figure 14–1 Application with Two Combo Boxes



Creating Combo Boxes

When creating a combo box, you must instantiate the `ComboBox` class and define the items as an observable list, just like other UI controls such as `ChoiceBox`, `ListView`, and `TableView`. Example 14–1 sets the items within a constructor.

Example 14–1 Creating a Combo Box with an Observable List

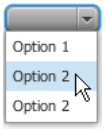
```
ObservableList<String> options =
    FXCollections.observableArrayList(
        "Option 1",
```

```
        "Option 2",
        "Option 3"
    );
final ComboBox comboBox = new ComboBox(options);
```

Another possibility is to create a combo box by using an empty constructor and call the `setItems` method on it, as follows: `comboBox.setItems(options);`

When the combo box is added to the application scene, it appears in the user interface as shown in Figure 14–2.

Figure 14–2 Combo Box with Three Items



At any time, you can supplement the list of items with new values. Example 14–2 implements this task by adding three more items to the comboBox control.

Example 14–2 Adding Items to a Combo Box

```
comboBox.getItems().addAll(
    "Option 4",
    "Option 5",
    "Option 6"
);
```

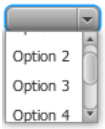
The `ComboBox` class provides handy properties and methods to use with combo boxes.

You can use the `setValue` method to specify the item selected in the combo box. When you call the `setValue` method on the `ComboBox` object, the selected item of the `selectionModel` property changes to this value even if the value is not in the combo box items list. If the items list then changes to include this value, the corresponding item becomes selected.

Similarly, you can obtain the value of the selected item by calling the `getValue` method. When a user selects an item, the selected item of the `selectionModel` property and the combo box `value` property are both updated to the new value.

You can also restrict the number of visible rows in the `ComboBox` drop down list when it is displayed. The following code line enables the display of three items for the comboBox control: `comboBox.setVisibleRowCount(3)` As the result of calling this method, the number of visible rows is limited to three, and a scroll bar appears (as shown in Figure 14–3).

Figure 14–3 Setting the Number of Visible Rows for a Combo Box



Although the `ComboBox` class has a generic notation and enables users to populate it with items of various types, do not use `Node` (or any subclass) as the type. Because the

scene graph concept implies that only one `Node` object can be in one place of the application scene, the selected item is removed from the `ComboBox` list of items. When the selection changes, the previously selected item returns to the list and the new selection is removed. To prevent this situation, use the cell factory mechanism and the solution described in the API documentation. The cell factory mechanism is particularly helpful when you need to change the initial behavior or appearance of the `ComboBox` object.

The ComboBoxSample application is designed to illustrate how to use combo boxes in a typical email interface. Example 14–3 creates a such an interface, in which two combo boxes are used to select the email recipient and the priority of the message.

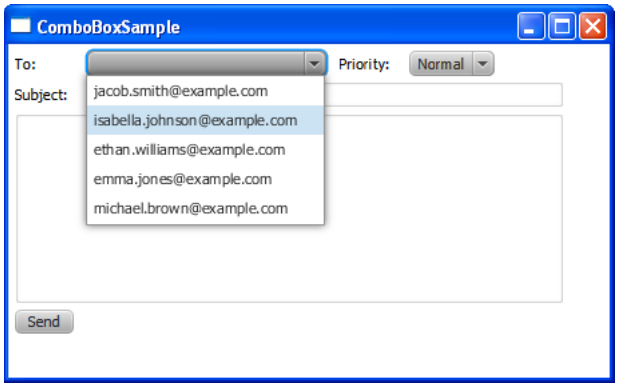
Example 14–3 Creating Combo Boxes and Adding Them to the Scene

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

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

    final Button button = new Button ("Send");
    final Label notification = new Label ();
    final TextField subject = new TextField("");
    final TextArea text = new TextArea ("");

    String address = " ";

    @Override public void start(Stage stage) {
        stage.setTitle("ComboBoxSample");
        Scene scene = new Scene(new Group(), 450, 250);

        final ComboBox emailComboBox = new ComboBox();
        emailComboBox.getItems().addAll(
            "jacob.smith@example.com",
            "isabella.johnson@example.com",
            "ethan.williams@example.com",
            "emma.jones@example.com",
            "michael.brown@example.com"
        );

        final ComboBox priorityComboBox = new ComboBox();
        priorityComboBox.getItems().addAll(
            "Highest",
            "High",
            "Normal",
            "Low",
            "Lowest"
        );

        priorityComboBox.setValue("Normal");

        GridPane grid = new GridPane();
        grid.setVgap(4);
```

```
        grid.setHgap(10);
        grid.setPadding(new Insets(5, 5, 5, 5));
        grid.add(new Label("To: "), 0, 0);
        grid.add(emailComboBox, 1, 0);
        grid.add(new Label("Priority: "), 2, 0);
        grid.add(priorityComboBox, 3, 0);
        grid.add(new Label("Subject: "), 0, 1);
        grid.add(subject, 1, 1, 3, 1);
        grid.add(text, 0, 2, 4, 1);
        grid.add(button, 0, 3);
        grid.add (notification, 1, 3, 3, 1);

        Group root = (Group)scene.getRoot();
        root.getChildren().add(grid);
        stage.setScene(scene);
        stage.show();
    }
}
```

Both combo boxes in Example 14–3 use the `getItems` and `addAll` methods to add items. When you compile and run this code, it produces the application window shown in Figure 14–4.

Figure 14–4 Email Recipient and Priority Combo Boxes



Editable Combo Boxes

Typically, email client applications enable users to both select recipients from the address book and type a new address. An editable combo box perfectly fits this task. Use the `setEditable(true)` method of the `ComboBox` class to make a combo box editable. With the `setPromptText` method, you can specify the text to appear in the combo box editing area when no selection is performed. Examine the modified code of the application in Example 14–4. The bold lines are the additions made to Example 14–3.

Example 14–4 Processing Newly Typed Values in an Editable Combo Box

```
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.scene.Group;
```

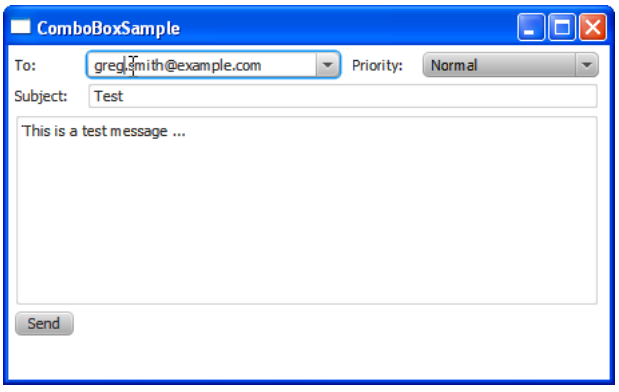
```
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

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

    final Button button = new Button ("Send");
    final Label notification = new Label ();
    final TextField subject = new TextField("");
    final TextArea text = new TextArea ("");

    String address = " ";

    @Override public void start(Stage stage) {
        stage.setTitle("ComboBoxSample");
        Scene scene = new Scene(new Group(), 450, 250);

        final ComboBox emailComboBox = new ComboBox();
        emailComboBox.getItems().addAll(
            "jacob.smith@example.com",
            "isabella.johnson@example.com",
            "ethan.williams@example.com",
            "emma.jones@example.com",
            "michael.brown@example.com"
        );
        emailComboBox.setPromptText("Email address");
        emailComboBox.setEditable(true);
        emailComboBox.valueProperty().addListener(new ChangeListener<String>() {
            @Override
          public void changed(ObservableValue ov, String t, String t1) {
                address = t1;
            }
        });

        final ComboBox priorityComboBox = new ComboBox();
        priorityComboBox.getItems().addAll(
            "Highest",
            "High",
            "Normal",
            "Low",
            "Lowest"
        );

        priorityComboBox.setValue("Normal");


        button.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent e) {
                if (emailComboBox.getValue() != null &&
                    !emailComboBox.getValue().toString().isEmpty()){
                        notification.setText("Your message was successfully sent"
                            + " to " + address);
                        emailComboBox.setValue(null);
                        if (priorityComboBox.getValue() != null &&
                            !priorityComboBox.getValue().toString().isEmpty()){
```

```
                priorityComboBox.setValue(null);
            }
            subject.clear();
            text.clear();
        }
        else {
            notification.setText("You have not selected a recipient!");
        }
    }
});

GridPane grid = new GridPane();
grid.setVgap(4);
grid.setHgap(10);
grid.setPadding(new Insets(5, 5, 5, 5));
grid.add(new Label("To: "), 0, 0);
grid.add(emailComboBox, 1, 0);
grid.add(new Label("Priority: "), 2, 0);
grid.add(priorityComboBox, 3, 0);
grid.add(new Label("Subject: "), 0, 1);
grid.add(subject, 1, 1, 3, 1);
grid.add(text, 0, 2, 4, 1);
grid.add(button, 0, 3);
grid.add (notification, 1, 3, 3, 1);

Group root = (Group)scene.getRoot();
root.getChildren().add(grid);
stage.setScene(scene);
stage.show();

    }
}
```

Besides the ability to edit `emailComboBox`, this code fragment implements event handling for this control. The newly typed or selected value is stored in the `address` variable. When users press the Send button, the notification containing the email address is shown.

Figure 14–5 captures the moment when a user is editing the email address of Jacob Smith and changing it to greg.smith@example.com.

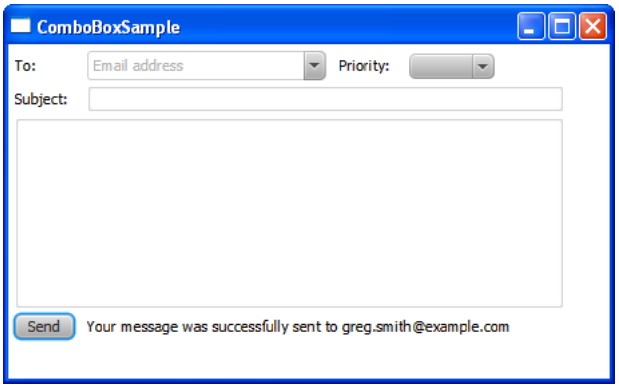
Figure 14–5 Editing an Email Address



When the Send button is pressed, all the controls return to their default states. The `clear` methods are called on the `TextField` and `TextArea` objects, and the `null`

value is set for the combo box selected items. Figure 14–6 shows the moment after the Send button is pressed.

Figure 14–6 User Interface After the Send Button Is Pressed



Applying Cell Factories to Combo Boxes

You can use the cell factory mechanism to alter the default behavior or appearance of a combo box. Example 14–5 creates a cell factory and applies it to the priority combo box to highlight priority types with special colors.

Example 14–5 Implementing a Cell Factory for the Priority Combo Box

```
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.GridPane;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
import javafx.util.Callback;

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

    final Button button = new Button ("Send");
    final Label notification = new Label ();
    final TextField subject = new TextField("");
    final TextArea text = new TextArea ("");

    String address = " ";

    @Override public void start(Stage stage) {
        stage.setTitle("ComboBoxSample");
        Scene scene = new Scene(new Group(), 450, 250);

        final ComboBox emailComboBox = new ComboBox();
```

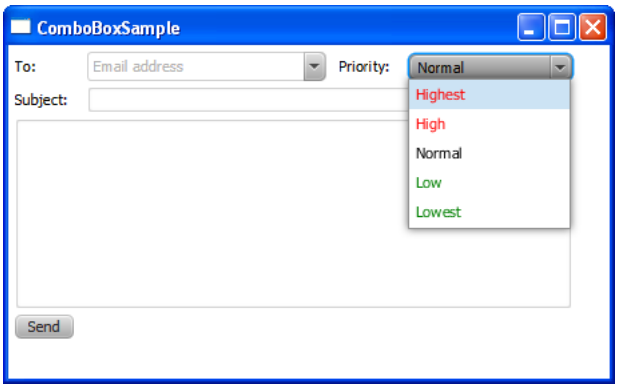
```
        emailComboBox.getItems().addAll(
            "jacob.smith@example.com",
            "isabella.johnson@example.com",
            "ethan.williams@example.com",
            "emma.jones@example.com",
            "michael.brown@example.com"
        );
        emailComboBox.setPromptText("Email address");
        emailComboBox.setEditable(true);
        emailComboBox.valueProperty().addListener(new ChangeListener<String>() {
            @Override public void changed(ObservableValue ov, String t, String t1)
{
                address = t1;
            }
        });

        final ComboBox priorityComboBox = new ComboBox();
        priorityComboBox.getItems().addAll(
            "Highest",
            "High",
            "Normal",
            "Low",
            "Lowest"
        );

        priorityComboBox.setValue("Normal");
        priorityComboBox.setCellFactory(
            new Callback<ListView<String>, ListCell<String>>() {
                @Override public ListCell<String> call(ListView<String> param) {
                    final ListCell<String> cell = new ListCell<String>() {
                        {
                            super.setPrefWidth(100);
                        }
                        @Override public void updateItem(String item,
                            boolean empty) {
                                super.updateItem(item, empty);
                                if (item != null) {
                                    setText(item);
                                    if (item.contains("High")) {
                                        setTextFill(Color.RED);
                                    }
                                    else if (item.contains("Low")){
                                        setTextFill(Color.GREEN);
                                    }
                                    else {
                                        setTextFill(Color.BLACK);
                                    }
                                }
                                else {
                                    setText(null);
                                }
                            }
                    };
                    return cell;
                }
        });


        button.setOnAction(new EventHandler<ActionEvent>() {
            @Override
```

```
            public void handle(ActionEvent e) {
                if (emailComboBox.getValue() != null &&
                    !emailComboBox.getValue().toString().isEmpty()){
                        notification.setText("Your message was successfully sent"
                            + " to " + address);
                        emailComboBox.setValue(null);
                        if (priorityComboBox.getValue() != null &&
                            !priorityComboBox.getValue().toString().isEmpty()){
                                priorityComboBox.setValue(null);
                            }
                        subject.clear();
                        text.clear();
                }
                else {
                    notification.setText("You have not selected a recipient!");
                }
            }
        });

        GridPane grid = new GridPane();
        grid.setVgap(4);
        grid.setHgap(10);
        grid.setPadding(new Insets(5, 5, 5, 5));
        grid.add(new Label("To: "), 0, 0);
        grid.add(emailComboBox, 1, 0);
        grid.add(new Label("Priority: "), 2, 0);
        grid.add(priorityComboBox, 3, 0);
        grid.add(new Label("Subject: "), 0, 1);
        grid.add(subject, 1, 1, 3, 1);
        grid.add(text, 0, 2, 4, 1);
        grid.add(button, 0, 3);
        grid.add (notification, 1, 3, 3, 1);

        Group root = (Group)scene.getRoot();
        root.getChildren().add(grid);
        stage.setScene(scene);
        stage.show();

    }
}
```

The cell factory produces `ListCell` objects. Every cell is associated with a single combo box item. The width of each combo box item is set through the `setPrefWidth` method. The `updateItem` method sets the red color for the High and Highest items, green color for the Low and Lowest items, and leaves the Normal item black.

Figure 14–7 shows the items of the priority combo box after the cell factory in Example 14–5 is applied.

Figure 14–7 Modified the Priority Combo Box



You can further enhance the appearance of the `ComboBox` control by applying CSS styles or visual effects.

Related API Documentation

- `ComboBox`
- `ComboBoxBase`
- `ListView`
- `ListCell`
- `Button`

15

Separator

This chapter explains how to use separator to organize UI components of you JavaFX applications.

The `Separator` class that is available in the JavaFX API represents a horizontal or vertical separator line. It serves to divide elements of the application user interface and does not produce any action. However, you can style it, apply visual effects to it, or even animate it. By default, the separator is horizontal. You can change its orientation by using the `setOrientation` method.

Creating a Separator

The code fragment in Example 15–1 creates one horizontal separator and one vertical separator.

Example 15–1 Vertical and Horizontal Separators

```
//Horizontal separator
Separator separator1 = new Separator();
//Vertical separator
Separator separator2 = new Separator();
separator2.setOrientation(Orientation.VERTICAL);
```

The `Separator` class is an extension of the `Node` class. Therefore, the separator inherits all the instance variables of the `Node` class.

Typically, separators are used to divide groups of the UI controls. Study the code fragment shown in Example 15–2. It separates the spring month checkboxes from the summer month checkboxes.

Example 15–2 Using a Separator Between Checkbox Categories

```
final String[] names = new String[]{"March", "April", "May",
    "June", "July", "August"};
final CheckBox[] cbs = new CheckBox[names.length];
final Separator separator = new Separator();
final VBox vbox = new VBox();

for (int i = 0; i < names.length; i++) {
    cbs[i] = new CheckBox(names[i]);
}

separator.setMaxWidth(40);
separator.setAlignment(Pos.CENTER_LEFT);
vbox.getChildren().addAll(cbs);
vbox.setSpacing(5);
```

```
vbox.getChildren().add(3, separator);
```

When this code fragment is added to an application, it produces the controls shown in Figure 15–1.

Figure 15–1 Checkboxes and a Separator

A separator occupies the full horizontal or vertical space allocated to it. The `setMaxWidth` method is applied to define a particular width. The `setValignment` method specifies the vertical position of the separator within the allocated layout space. Similarly, you can set the horizontal position of the separator line by applying the `setHalignment` method.

In Example 15–2, the separator is added to the vertical box by using a dedicated method `add(index, node)`. You can use this approach in your application to include separators after the UI is created or when the UI is dynamically changed.

Adding Separators to the UI of Your Application

As previously mentioned, separators can be used to divide groups of UI controls. You can also use them to structure a user interface. Consider the task of rendering the weather forecast data shown in Figure 15–2.

Figure 15–2 Structuring Weather Forecast Data with Separators

For the application shown in Figure 15–2, separators are used to divide `Label` and `ImageView` objects. Study the source code of this application shown in Example 15–3.

Example 15–3 Using Separators in a Weather Forecast Application

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Orientation;
import javafx.geometry.VPos;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.GridPane;
```

```
import javafx.scene.text.Font;
import javafx.stage.Stage;

public class Main extends Application {

    Label caption = new Label("Weather Forecast");
    Label friday = new Label("Friday");
    Label saturday = new Label("Saturday");
    Label sunday = new Label("Sunday");

    @Override
    public void start(Stage stage) {
        Group root = new Group();
        Scene scene = new Scene(root, 500, 300);
        stage.setScene(scene);
        stage.setTitle("Separator Sample");

        GridPane grid = new GridPane();
        grid.setPadding(new Insets(10, 10, 10, 10));
        grid.setVgap(2);
        grid.setHgap(5);

        scene.setRoot(grid);

        Image cloudImage = new Image(getClass().getResourceAsStream("cloud.jpg"));
        Image sunImage = new Image(getClass().getResourceAsStream("sun.jpg"));

        caption.setFont(Font.font("Verdana", 20));
        GridPane.setConstraints(caption, 0, 0);
        GridPane.setColumnSpan(caption, 8);
        grid.getChildren().add(caption);

        final Separator sepHor = new Separator();
        sepHor.setValignment(VPos.CENTER);
        GridPane.setConstraints(sepHor, 0, 1);
        GridPane.setColumnSpan(sepHor, 7);
        grid.getChildren().add(sepHor);

        friday.setFont(Font.font("Verdana", 18));
        GridPane.setConstraints(friday, 0, 2);
        GridPane.setColumnSpan(friday, 2);
        grid.getChildren().add(friday);

        final Separator sepVert1 = new Separator();
        sepVert1.setOrientation(Orientation.VERTICAL);
        sepVert1.setValignment(VPos.CENTER);
        sepVert1.setPrefHeight(80);
        GridPane.setConstraints(sepVert1, 2, 2);
        GridPane.setRowSpan(sepVert1, 2);
        grid.getChildren().add(sepVert1);

        saturday.setFont(Font.font("Verdana", 18));
        GridPane.setConstraints(saturday, 3, 2);
        GridPane.setColumnSpan(saturday, 2);
        grid.getChildren().add(saturday);

        final Separator sepVert2 = new Separator();
        sepVert2.setOrientation(Orientation.VERTICAL);
        sepVert2.setValignment(VPos.CENTER);
        sepVert2.setPrefHeight(80);
```

```
            GridPane.setConstraints(sepVert2, 5, 2);
            GridPane.setRowSpan(sepVert2, 2);
            grid.getChildren().add(sepVert2);

            sunday.setFont(Font.font("Verdana", 18));
            GridPane.setConstraints(sunday, 6, 2);
            GridPane.setColumnSpan(sunday, 2);
            grid.getChildren().add(sunday);

            final ImageView cloud = new ImageView(cloudImage);
            GridPane.setConstraints(cloud, 0, 3);
            grid.getChildren().add(cloud);

            final Label t1 = new Label("16");
            t1.setFont(Font.font("Verdana", 20));
            GridPane.setConstraints(t1, 1, 3);
            grid.getChildren().add(t1);

            final ImageView sun1 = new ImageView(sunImage);
            GridPane.setConstraints(sun1, 3, 3);
            grid.getChildren().add(sun1);

            final Label t2 = new Label("18");
            t2.setFont(Font.font("Verdana", 20));
            GridPane.setConstraints(t2, 4, 3);
            grid.getChildren().add(t2);

            final ImageView sun2 = new ImageView(sunImage);
            GridPane.setConstraints(sun2, 6, 3);
            grid.getChildren().add(sun2);

            final Label t3 = new Label("20");
            t3.setFont(Font.font("Verdana", 20));
            GridPane.setConstraints(t3, 7, 3);
            grid.getChildren().add(t3);

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

This application uses both horizontal and vertical separators and makes the separators span rows and columns in the `GridPane` container. In your application, you can also set the preferred length for a separator (width for a horizontal separator and height for a vertical separator), so that it can change dynamically when the user interface resizes. You can also alter the visual appearance of a separator by applying the CSS classes available for `Separator` objects.

Styling Separators

To apply the same style to all the separators in Example 15–3, you create CSS file (for example, controlStyle.css) and save this file in the same package as the main class of your application. Example 15–4 demonstrates the CSS classes that you can add to the controlStyle file.

Example 15–4 Using CSS Classes to Style Separators

```
/*controlStyle.css */

.separator{
    -fx-background-color: #e79423;
    -fx-background-radius: 2;
}
```

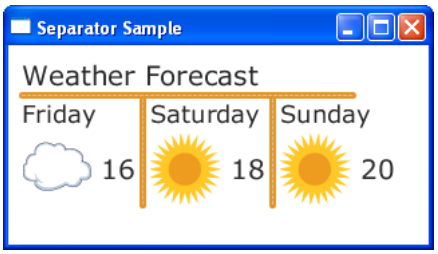
You can enable the separator style in the application through the `getStylesheets` method of the `Scene` class, as shown in Example 15–5.

Example 15–5 Enabling Style Sheets in a JavaFX Application

```
scene.getStylesheets().add("separatorsample/controlStyle.css");
```

Figure 15–3 shows how the separators in the weather forecast look when the modified application is compiled and run.

Figure 15–3 Styled Separators



Related API Documentation

- `Separator`
- JavaFX CSS Specification

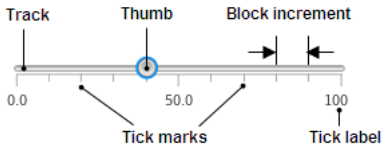
16

Slider

In this chapter, you learn how to use sliders in your JavaFX applications to display and interact with a range of numeric values.

The `Slider` control consists of a track and a draggable thumb. It can also include tick marks and tick labels that indicate numeric values of the range. Figure 16–1 shows a typical slider and indicates its main elements.

Figure 16–1 Elements of a Slider



Creating a Slider

Take a moment to review the code fragment in Example 16–1 that produces a slider shown in Figure 16–1.

Example 16–1 Creating a Slider

```
Slider slider = new Slider();
slider.setMin(0);
slider.setMax(100);
slider.setValue(40);
slider.setShowTickLabels(true);
slider.setShowTickMarks(true);
slider.setMajorTickUnit(50);
slider.setMinorTickCount(5);
slider.setBlockIncrement(10);
```

The `setMin` and `setMax` methods define, respectively, the minimum and the maximum numeric values represented by the slider. The `setValue` method specifies the current value of the slider, which is always less than the maximum value and more than the minimum value. Use this method to define the position of the thumb when the application starts.

Two Boolean methods, `setShowTickMarks` and `setShowTickLabels`, define the visual appearance of the slider. In Example 16–1, the marks and labels are enabled. Additionally, the unit distance between major tick marks is set to 50, and the number of minor ticks between any two major ticks is specified as 5. You can assign the

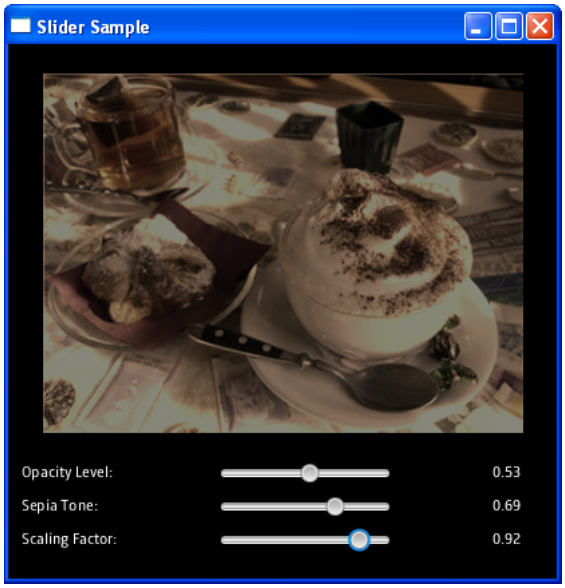
`setSnapToTicks` method to `true` to keep the slider's value aligned with the tick marks.

The `setBlockIncrement` method defines the distance that the thumb moves when a user clicks on the track. In Example 16–1, this value is 10, which means that each time a user clicks on the track, the thumb moves 10 units toward the click location.

Using Sliders in Graphical Applications

Now examine Figure 16–2. This application uses three sliders to edit rendering characteristics of a picture. Each slider adjusts a particular visual characteristic: opacity level, sepia tone value, or scaling factor.

Figure 16–2 Three Sliders



Example 16–2 shows the source code of this application.

Example 16–2 Slider Sample

```
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.geometry.Insets;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.Slider;
import javafx.scene.effect.SepiaTone;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.GridPane;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class Main extends Application {

    final Slider opacityLevel = new Slider(0, 1, 1);
```

```
final Slider sepiaTone = new Slider(0, 1, 1);
final Slider scaling = new Slider (0.5, 1, 1);
final Image image  = new Image(getClass().getResourceAsStream(
    "cappuccino.jpg")
);

final Label opacityCaption = new Label("Opacity Level:");
final Label sepiaCaption = new Label("Sepia Tone:");
final Label scalingCaption = new Label("Scaling Factor:");

final Label opacityValue = new Label(
    Double.toString(opacityLevel.getValue()));
final Label sepiaValue = new Label(
    Double.toString(sepiaTone.getValue()));
final Label scalingValue = new Label(
    Double.toString(scaling.getValue()));

final static Color textColor = Color.WHITE;
final static SepiaTone sepiaEffect = new SepiaTone();

@Override
public void start(Stage stage) {
    Group root = new Group();
    Scene scene = new Scene(root, 600, 400);
    stage.setScene(scene);
    stage.setTitle("Slider Sample");
    scene.setFill(Color.BLACK);

    GridPane grid = new GridPane();
    grid.setPadding(new Insets(10, 10, 10, 10));
    grid.setVgap(10);
    grid.setHgap(70);

    final ImageView cappuccino = new ImageView (image);
    cappuccino.setEffect(sepiaEffect);
    GridPane.setConstraints(cappuccino, 0, 0);
    GridPane.setColumnSpan(cappuccino, 3);
    grid.getChildren().add(cappuccino);
    scene.setRoot(grid);

    opacityCaption.setTextFill(textColor);
    GridPane.setConstraints(opacityCaption, 0, 1);
    grid.getChildren().add(opacityCaption);


    opacityLevel.valueProperty().addListener(new ChangeListener<Number>() {
        public void changed(ObservableValue<? extends Number> ov,
            Number old_val, Number new_val) {
                cappuccino.setOpacity(new_val.doubleValue());
                opacityValue.setText(String.format("%.2f", new_val));
        }
    });

    GridPane.setConstraints(opacityLevel, 1, 1);
    grid.getChildren().add(opacityLevel);

    opacityValue.setTextFill(textColor);
    GridPane.setConstraints(opacityValue, 2, 1);
    grid.getChildren().add(opacityValue);
```

```
            sepiaCaption.setTextFill(textColor);
            GridPane.setConstraints(sepiaCaption, 0, 2);
            grid.getChildren().add(sepiaCaption);

            sepiaTone.valueProperty().addListener(new ChangeListener<Number>() {
                public void changed(ObservableValue<? extends Number> ov,
                    Number old_val, Number new_val) {
                        sepiaEffect.setLevel(new_val.doubleValue());
                        sepiaValue.setText(String.format("%.2f", new_val));
                }
            });
            GridPane.setConstraints(sepiaTone, 1, 2);
            grid.getChildren().add(sepiaTone);

            sepiaValue.setTextFill(textColor);
            GridPane.setConstraints(sepiaValue, 2, 2);
            grid.getChildren().add(sepiaValue);

            scalingCaption.setTextFill(textColor);
            GridPane.setConstraints(scalingCaption, 0, 3);
            grid.getChildren().add(scalingCaption);

            scaling.valueProperty().addListener(new ChangeListener<Number>() {
                public void changed(ObservableValue<? extends Number> ov,
                    Number old_val, Number new_val) {
                        cappuccino.setScaleX(new_val.doubleValue());
                        cappuccino.setScaleY(new_val.doubleValue());
                        scalingValue.setText(String.format("%.2f", new_val));
                }
            });
            GridPane.setConstraints(scaling, 1, 3);
            grid.getChildren().add(scaling);

            scalingValue.setTextFill(textColor);
            GridPane.setConstraints(scalingValue, 2, 3);
            grid.getChildren().add(scalingValue);

            stage.show();
        }

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

The opacity property of the `ImageView` object changes in accordance with the value of the first slider, named opacityLevel. The level of the `SepiaTone` effect changes when the value of the sepiaTone slider changes. The third slider defines the scaling factor for the picture by passing to the `setScaleX` and `setScaleY` methods the current value of the slider.

The code fragment in Example 16–3 demonstrates the methods that convert the double value returned by the `getValue` method of the `Slider` class into `String`. It also applies formatting to render the slider's value as a float number with two digits after the point.

Example 16–3 Formatting the Rendered Slider's Value

```
scalingValue.setText((Double.toString(value)).format("%.2f", value));
```

The next step to enhance the look and feel of a slider is to apply visual effects or CSS styles to it.

Related API Documentation

- `Slider`
- `SepiaTone`

17

Progress Bar and Progress Indicator

In this chapter, you learn about the progress indicator and progress bar, the UI controls that visualize progress of any operations in your JavaFX applications.

The `ProgressIndicator` class and its direct subclass `ProgressBar` provide the capabilities to indicate that a particular task is processing and to detect how much of this work has been already done. While the `ProgressBar` class visualizes the progress as a completion bar, the `ProgressIndicator` class visualizes the progress in the form of a dynamically changing pie chart, as shown in Figure 17–1.

Figure 17–1 Progress Bar and Progress Indicator

Creating Progress Controls

Use the code fragment in Example 17–1 to insert the progress controls in your JavaFX application.

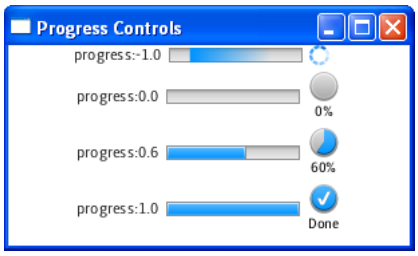
Example 17–1 Implementing the Progress Bar and Progress Indicator

```
ProgressBar pb = new ProgressBar(0.6);
ProgressIndicator pi = new ProgressIndicator(0.6);
```

You can also create the progress controls without parameters by using an empty constructor. In that case, you can assign the value by using the `setProgress` method.

Sometimes an application cannot determine the full completion time of a task. In that case, progress controls remain in indeterminate mode until the length of the task is determined. Figure 17–2 shows different states of the progress controls depending on their progress variable value.

Figure 17–2 Various States of Progress Controls



Example 17–2 shows the source code of the application shown in Figure 17–2.

Example 17–2 Enabling Different States of Progress Controls

```
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.ProgressBar;
import javafx.scene.control.ProgressIndicator;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class Main  extends Application {

final Float[] values = new Float[] {-1.0f, 0f, 0.6f, 1.0f};
final Label [] labels = new Label[values.length];
final ProgressBar[] pbs = new ProgressBar[values.length];
final ProgressIndicator[] pins = new ProgressIndicator[values.length];
final HBox hbs [] = new HBox [values.length];

    @Override
    public void start(Stage stage) {
        Group root = new Group();
        Scene scene = new Scene(root, 300, 150);
        scene.getStylesheets().add("progresssample/Style.css");
        stage.setScene(scene);
        stage.setTitle("Progress Controls");


        for (int i = 0; i < values.length; i++) {
            final Label label = labels[i] = new Label();
            label.setText("progress:" + values[i]);

            final ProgressBar pb = pbs[i] = new ProgressBar();
            pb.setProgress(values[i]);

            final ProgressIndicator pin = pins[i] = new ProgressIndicator();
            pin.setProgress(values[i]);
            final HBox hb = hbs[i] = new HBox();
            hb.setSpacing(5);
            hb.setAlignment(Pos.CENTER);
            hb.getChildren().addAll(label, pb, pin);
        }

        final VBox vb = new VBox();
```

```
        vb.setSpacing(5);
        vb.getChildren().addAll(hbs);
        scene.setRoot(vb);
        stage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

A positive value of the progress variable between 0 and 1 indicates the percentage of progress. For example, 0.4 corresponds to 40%. A negative value for this variable indicates that the progress is in the indeterminate mode. Use the `isIndeterminate` method to check whether the progress control is in the indeterminate mode.

Indicating Progress in Your User Interface

Example 17–2 was initially simplified to render all the possible states of the progress controls. In real-world applications, the progress value can be obtained through the value of other UI elements.

Study the code in Example 17–3 to learn how set values for the progress bar and progress indicator based on the slider position.

Example 17–3 Receiving the Progress Value from a Slider

```
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.geometry.Pos;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.ProgressBar;
import javafx.scene.control.ProgressIndicator;
import javafx.scene.control.Slider;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class Main  extends Application {

    @Override
    public void start(Stage stage) {
        Group root = new Group();
        Scene scene = new Scene(root);
        stage.setScene(scene);
        stage.setTitle("Progress Controls");

        final Slider slider = new Slider();
        slider.setMin(0);
        slider.setMax(50);

        final ProgressBar pb = new ProgressBar(0);
        final ProgressIndicator pi = new ProgressIndicator(0);

        slider.valueProperty().addListener(new ChangeListener<Number>() {
            public void changed(ObservableValue<? extends Number> ov,
                Number old_val, Number new_val) {
                pb.setProgress(new_val.doubleValue()/50);
                pi.setProgress(new_val.doubleValue()/50);
```

```
            }
        });

        final HBox hb = new HBox();
        hb.setSpacing(5);
        hb.setAlignment(Pos.CENTER);
        hb.getChildren().addAll(slider, pb, pi);
        scene.setRoot(hb);
        stage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```

When you compile and run this application, it produces the window shown in Figure 17–3.

Figure 17–3 Indicating the Progress Set by a Slider

An `ChangeListener` object determines if the slider's value is changed and computes the progress for the progress bar and progress indicator so that the values of the progress controls are in the range of 0.0 to 1.0.

Related API Documentation.

- `ProgressBar`
- `ProgressIndicator`

18

Hyperlink

This chapter tells about the `Hyperlink` control used to format text as a hyperlink.

The `Hyperlink` class represents another type of `Labeled` control. Figure 18–1 demonstrates three states of the default hyperlink implementation.

Figure 18–1 Three States of a Hyperlink Control

Creating a Hyperlink

The code fragment that produces a hyperlink is shown in Example 18–1.

Example 18–1 Typical Hyperlink

```
Hyperlink link = new Hyperlink();
link.setText("http://example.com");
link.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent e) {
        System.out.println("This link is clicked");
    }
});
```

The `setText` instance method defines the text caption for the hyperlink. Because the `Hyperlink` class is an extension of the `Labeled` class, you can set a specific font and text fill for the hyperlink caption. The `setOnAction` method sets the specific action, which is called whenever the hyperlink is clicked, similar to how this method works for the `Button` control. In Example 18–1, this action is limited to printing the string. However, in your application, you might want to implement more common tasks.

Linking the Local Content

The application in Figure 18–2 renders images from the local directory.

Figure 18–2 Viewing Images

Review the source code of this application shown in Example 18–2.

Example 18–2 Using Hyperlinks to VIew Images

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.*;
import javafx.scene.control.*;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class Main extends Application {

    final static String[] imageFiles = new String[]{
        "product.png",
        "education.png",
        "partners.png",
        "support.png"
    };
    final static String[] captions = new String[]{
        "Products",
        "Education",
        "Partners",
        "Support"
    };
    final ImageView selectedImage = new ImageView();
    final ScrollPane list = new ScrollPane();
    final Hyperlink[] hpls = new Hyperlink[captions.length];
    final Image[] images = new Image[imageFiles.length];

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

    @Override
    public void start(Stage stage) {
        Scene scene = new Scene(new Group());
        stage.setTitle("Hyperlink Sample");
        stage.setWidth(300);
        stage.setHeight(200);

        selectedImage.setLayoutX(100);
        selectedImage.setLayoutY(10);

        for (int i = 0; i < captions.length; i++) {
            final Hyperlink hpl = hpls[i] = new Hyperlink(captions[i]);
```

```
            final Image image = images[i] = new Image(
                getClass().getResourceAsStream(imageFiles[i])
            );
            hpl.setOnAction(new EventHandler<ActionEvent>() {
                @Override
                public void handle(ActionEvent e) {
                    selectedImage.setImage(image);
                }
            });
        }

        final Button button = new Button("Refresh links");
        button.setOnAction(new EventHandler<ActionEvent>() {
                @Override
                public void handle(ActionEvent e) {
                    for (int i = 0; i < captions.length; i++) {
                        hpls[i].setVisited(false);
                        selectedImage.setImage(null);
                    }
                }
            });

        VBox vbox = new VBox();
        vbox.getChildren().addAll(hpls);
        vbox.getChildren().add(button);
        vbox.setSpacing(5);

        ((Group) scene.getRoot()).getChildren().addAll(vbox, selectedImage);
        stage.setScene(scene);
        stage.show();
    }
}
```

This application creates four `Hyperlink` objects within a `for` loop. The `setOnAction` method called on each hyperlink defines the behavior when a user clicks a particular hyperlink. In that case, the corresponding image from the `images` array is set for the `selectedImage` variable.

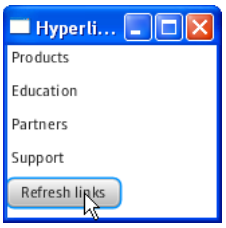
When a user clicks a hyperlink, it becomes visited. You can use the `setVisited` method of the `Hyperlink` class to refresh the links. The code fragment in Example 18–3 accomplishes this task.

Example 18–3 Refreshing the HyperlInks

```
final Button button = new Button("Refresh links");
button.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent e) {
       for (int i = 0; i < captions.length; i++) {
           hpls[i].setVisited(false);
           selectedImage.setImage(null);
       }
    }
});
```

When clicked, the Refresh Links button brings all the hyperlinks to the unvisited state, as show in Figure 18–3.

Figure 18–3 Unvisited Hyperlinks



Because the `Hyperlink` class is an extension of the `Labeled` class, you can specify not only a text caption but also an image. The application provided in the next section uses both a text caption and an image to create hyperlinks and to load remote HTML pages.

Linking the Remote Content

You can render HTML content in your JavaFX application by embedding the `WebView` browser in the application scene. The `WebView` component provides basic web page browsing functionality. It renders web pages and supports user interaction such as navigating links and executing JavaScript commands.

Study the source code of the application in Example 18–4. It creates four hyperlinks with text captions and images. When a hyperlink is clicked, the corresponding value is passed as a URL to the embedded browser.

Example 18–4 Loading Remote Web Pages

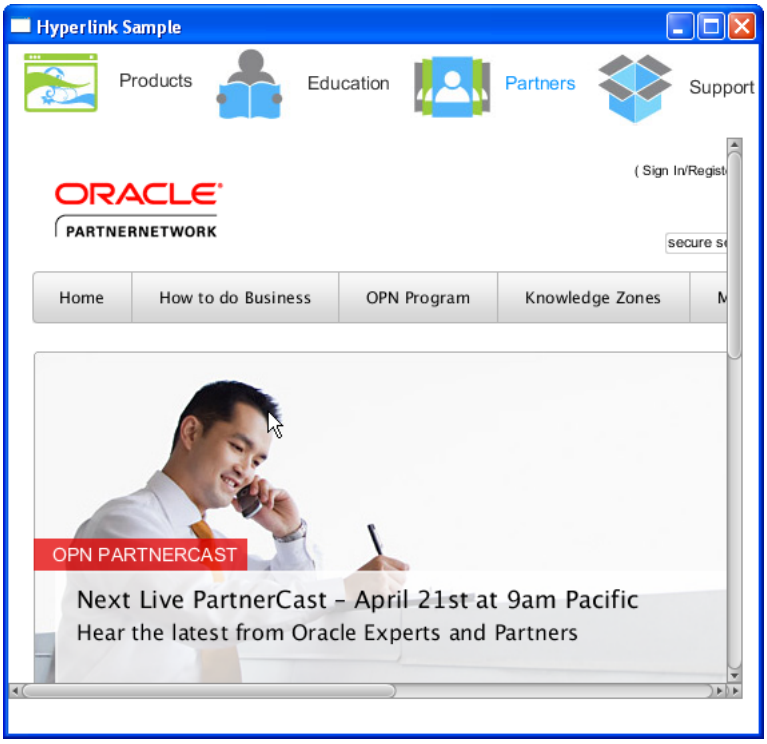
```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.*;
import javafx.scene.control.*;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Priority;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.scene.web.WebEngine;
import javafx.scene.web.WebView;
import javafx.stage.Stage;

public class Main extends Application {

    final static String[] imageFiles = new String[]{
        "product.png",
        "education.png",
        "partners.png",
        "support.png"
    };
    final static String[] captions = new String[]{
        "Products",
        "Education",
        "Partners",
        "Support"
    };
```

```
    final static String[] urls = new String[]{
        "http://www.oracle.com/us/products/index.html",
        "http://education.oracle.com/",
        "http://www.oracle.com/partners/index.html",
        "http://www.oracle.com/us/support/index.html"
    };

    final ImageView selectedImage = new ImageView();
    final Hyperlink[] hpls = new Hyperlink[captions.length];
    final Image[] images = new Image[imageFiles.length];

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

    @Override
    public void start(Stage stage) {
        VBox vbox = new VBox();
        Scene scene = new Scene(vbox);
        stage.setTitle("Hyperlink Sample");
        stage.setWidth(570);
        stage.setHeight(550);

        selectedImage.setLayoutX(100);
        selectedImage.setLayoutY(10);

        final WebView browser = new WebView();
        final WebEngine webEngine = browser.getEngine();

        for (int i = 0; i < captions.length; i++) {
            final Hyperlink hpl = hpls[i] = new Hyperlink(captions[i]);

            final Image image = images[i] =
                    new Image(getClass().getResourceAsStream(imageFiles[i]));
            hpl.setGraphic(new ImageView (image));
            hpl.setFont(Font.font("Arial", 14));
            final String url = urls[i];

            hpl.setOnAction(new EventHandler<ActionEvent>() {
                @Override
                public void handle(ActionEvent e) {
                    webEngine.load(url);
                }
            });
        }

        HBox hbox = new HBox();
        hbox.getChildren().addAll(hpls);

        vbox.getChildren().addAll(hbox, browser);
        VBox.setVgrow(browser, Priority.ALWAYS);

        stage.setScene(scene);
        stage.show();
    }
}
```

The hyperlinks are created within a `for` loop similar to the one in Example 18–2. The action set for a hyperlink passes the corresponding URL from the `urls` array to the `WebEngine` object of the embedded browser.

When you compile and run this application, it produces the window shown in Figure 18–4.

Figure 18–4 Loading Pages from the Oracle Corporate Site



Related API Documentation

- `Hyperlink`
- `Labeled`
- `WebView`
- `WebEngine`

19

HTML Editor

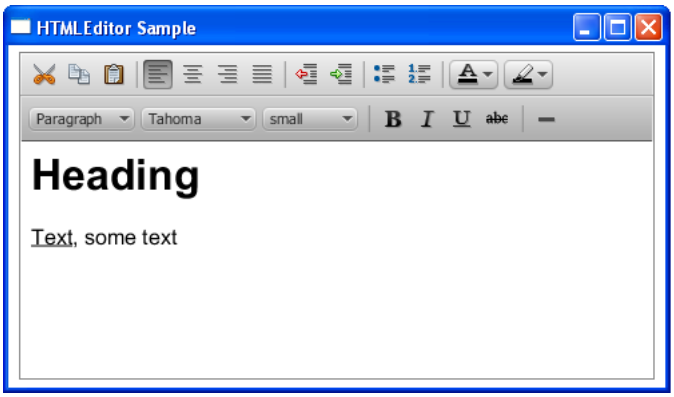
In this chapter, you learn how to edit text in your JavaFX applications by using the embedded HTML editor.

The `HTMLEditor` control is a full functional rich text editor. Its implementation is based on the document editing feature of HTML5 and includes the following editing functions:

- Text formatting including bold, italic, underline, and strike though styles
- Paragraph settings such as format, font family, and font size
- Foreground and background colors
- Text indent
- Bulleted and numbered lists
- Text alignment
- Adding a horizontal rule
- Copying and pasting text fragments

Figure 19–1 shows a rich text editor added to a JavaFX application.

Figure 19–1 HTML Editor



The `HTMLEditor` class presents the editing content in the form of an HTML string. For example, the content typed in the editor in Figure 19–1 is presented by the following string: "`<html><head></head><body contenteditable="true"><h1>Heading</h1><div><u>Text</u>, some text</div></body></html>`."

Because the `HTMLEditor` class is an extension of the `Node` class, you can apply visual effects or transformations to its instances.

Adding an HTML Editor

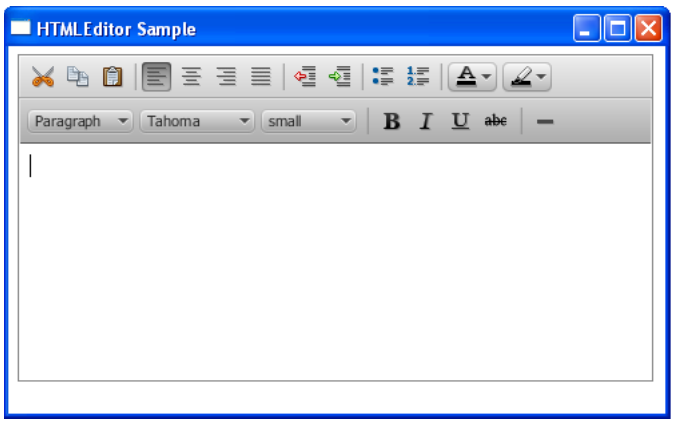
Like any other UI control, the `HTMLEditor` component must be added the scene so that it can appear in your application. You can add it directly to the scene as shown in Example 19–1 or through a layout container as done in other examples.

Example 19–1 Adding an HTML Editor to a JavaFX Application

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

public class HTMLEditorSample extends Application {

    @Override
    public void start(Stage stage) {
        stage.setTitle("HTMLEditor Sample");
        stage.setWidth(400);
        stage.setHeight(300);
        final HTMLEditor htmlEditor = new HTMLEditor();
        htmlEditor.setPrefHeight(245);
        Scene scene = new Scene(htmlEditor);
        stage.setScene(scene);
        stage.show();
    }

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

Compiling and running this code fragment produces the window shown in Figure 19–2.

Figure 19–2 Initial View of the HTMLEditor Component



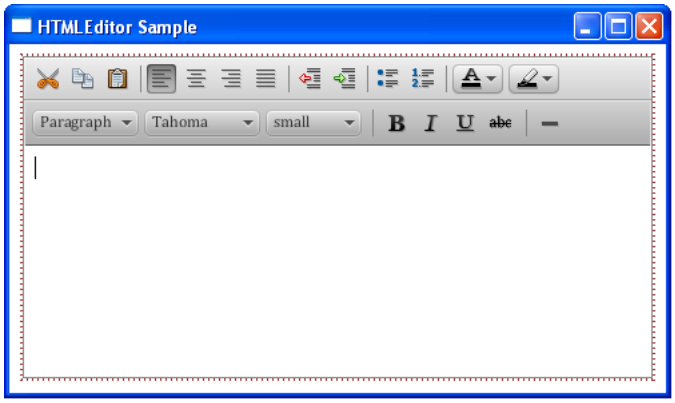
The formatting toolbars are provided in the implementation of the component. You cannot toggle their visibility. However, you still can customize the appearance of the editor by applying CSS style, as shown in Example 19–2.

Example 19–2 Styling the HTMLEditor

```
htmlEditor.setStyle(
    "-fx-font: 12 cambria;"
    + "-fx-border-color: brown; "
    + "-fx-border-style: dotted;"
    + "-fx-border-width: 2;"
);
```

When this code line is added to Example 19–1, the editor changes, as shown in Figure 19–3.

Figure 19–3 Alternative View of the HTMLEditor Component



The applied style changes the border of the component and the font of the formatting toolbars.

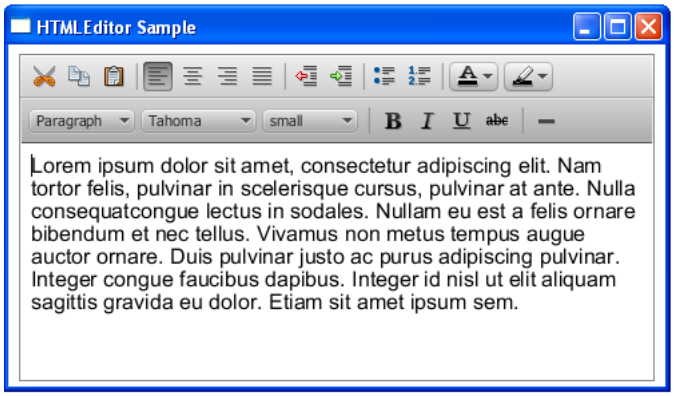
The `HTMLEditor` class provides a method to define the content that appears in the editing area when the application starts. Use the `setHtmlText` method, as shown in Example 19–3 to set the initial text for the editor.

Example 19–3 Setting the Text Content

```
private final String INITIAL_TEXT = "<html><body>Lorem ipsum dolor sit "
    + "amet, consectetur adipiscing elit. Nam tortor felis, pulvinar "
    + "in scelerisque cursus, pulvinar at ante. Nulla consequat"
    + "congue lectus in sodales. Nullam eu est a felis ornare "
    + "bibendum et nec tellus. Vivamus non metus tempus augue auctor "
    + "ornare. Duis pulvinar justo ac purus adipiscing pulvinar. "
    + "Integer congue faucibus dapibus. Integer id nisl ut elit "
    + "aliquam sagittis gravida eu dolor. Etiam sit amet ipsum "
    + "sem.</body></html>";

htmlEditor.setHtmlText(INITIAL_TEXT);
```

Figure 19–4 demonstrates the text editor with the text set by using the `setHTMLText` method.

Figure 19–4 HTMLEditor with the Predefined Text Content



You can use the HTML tags in this string to apply specific formatting for the initially rendered content.

Using an HTML Editor to Build the User Interface

You can use the `HTMLEditor` control to implement typical user interfaces (UIs) in your JavaFX applications. For example, you can implement instant messenger services, email clients, or even content management systems.

Example 19–4 presents the user interface of a message composing window that you can find in many email client applications.

Example 19–4 HTMLEditor Added to the Email Client UI

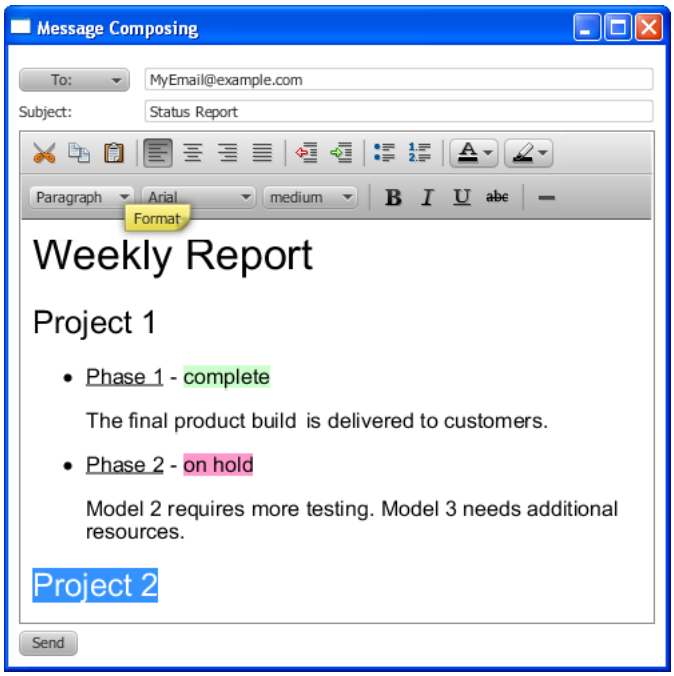
```
import javafx.application.Application;
import javafx.collections.FXCollections;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.VBox;
import javafx.scene.web.HTMLEditor;
import javafx.stage.Stage;

public class HTMLEditorSample extends Application {

    @Override
    public void start(Stage stage) {
        stage.setTitle("Message Composing");
        stage.setWidth(500);
        stage.setHeight(500);
        Scene scene = new Scene(new Group());

        final VBox root = new VBox();
        root.setPadding(new Insets(8, 8, 8, 8));
        root.setSpacing(5);
        root.setAlignment(Pos.BOTTOM_LEFT);

        final GridPane grid = new GridPane();
        grid.setVgap(5);
```

```
        grid.setHgap(10);

        final ChoiceBox sendTo =
            new ChoiceBox(FXCollections.observableArrayList(
                "To:", "Cc:", "Bcc:")
        );

        sendTo.setPrefWidth(100);
        GridPane.setConstraints(sendTo, 0, 0);
        grid.getChildren().add(sendTo);

        final TextField tbTo = new TextField();
        tbTo.setPrefWidth(400);
        GridPane.setConstraints(tbTo, 1, 0);
        grid.getChildren().add(tbTo);

        final Label subjectLabel = new Label("Subject:");
        GridPane.setConstraints(subjectLabel, 0, 1);
        grid.getChildren().add(subjectLabel);

        final TextField tbSubject = new TextField();
        tbTo.setPrefWidth(400);
        GridPane.setConstraints(tbSubject, 1, 1);
        grid.getChildren().add(tbSubject);

        root.getChildren().add(grid);

        final HTMLEditor htmlEditor = new HTMLEditor();
        htmlEditor.setPrefHeight(370);

        root.getChildren().addAll(htmlEditor, new Button("Send"));

        final Label htmlLabel = new Label();
        htmlLabel.setWrapText(true);

        scene.setRoot(root);
        stage.setScene(scene);
        stage.show();
    }

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

The user interface includes a choice box to select a type of recipient, two text fields to enter the email address and the subject of the message, a label to indicate the subject field, the editor, and the Send button.

The UI controls are arranged on the application scene by using the `Grid` and `VBox` layout containers. When you compile and run this application, the window shown in Figure 19–5 shows the output of this application when a user is composing a weekly report.

Figure 19–5 Email Client User Interface



You can set the specific width and height values for the `HTMLEditor` object by calling the `setPrefWidth` or `setPrefHeight` methods, or you can leave its width and height unspecified. Example 19–4 specifies the height of the component. Its width is defined by the `VBox` layout container. When the text content exceeds the height of the editing area, the vertical scroll bar appears.

Obtaining HTML Content

With the JavaFX `HTMLEditor` control, you can edit text and set the initial content. In addition, you can obtain the entered and edited text in HTML format. The application shown in Example 19–5 implements this task.

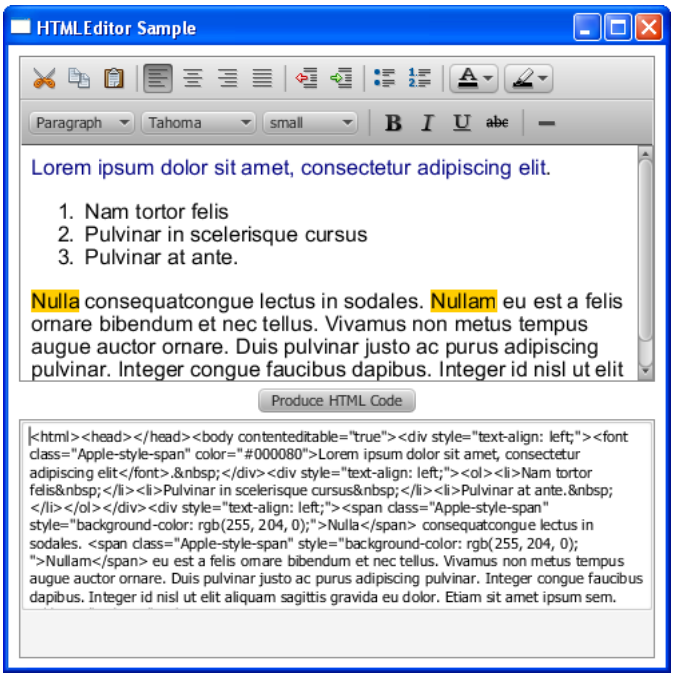
Example 19–5 Retrieving HTML Code

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.VBox;
import javafx.scene.web.HTMLEditor;
import javafx.stage.Stage;

public class HTMLEditorSample extends Application {
    private final String INITIAL_TEXT = "Lorem ipsum dolor sit "
            + "amet, consectetur adipiscing elit. Nam tortor felis, pulvinar "
            + "in scelerisque cursus, pulvinar at ante. Nulla consequat"
            + "congue lectus in sodales. Nullam eu est a felis ornare "
            + "bibendum et nec tellus. Vivamus non metus tempus augue auctor "
            + "ornare. Duis pulvinar justo ac purus adipiscing pulvinar. "
```

```
            + "Integer congue faucibus dapibus. Integer id nisl ut elit "
            + "aliquam sagittis gravida eu dolor. Etiam sit amet ipsum "
            + "sem.";

    @Override
    public void start(Stage stage) {
        stage.setTitle("HTMLEditor Sample");
        stage.setWidth(500);
        stage.setHeight(500);
        Scene scene = new Scene(new Group());

        VBox root = new VBox();
        root.setPadding(new Insets(8, 8, 8, 8));
        root.setSpacing(5);
        root.setAlignment(Pos.BOTTOM_LEFT);

        final HTMLEditor htmlEditor = new HTMLEditor();
        htmlEditor.setPrefHeight(245);
        htmlEditor.setHtmlText(INITIAL_TEXT);

        final TextArea htmlCode = new TextArea();
        htmlCode.setWrapText(true);

        ScrollPane scrollPane = new ScrollPane();
        scrollPane.getStyleClass().add("noborder-scroll-pane");
        scrollPane.setContent(htmlCode);
        scrollPane.setFitToWidth(true);
        scrollPane.setPrefHeight(180);

        Button showHTMLButton = new Button("Produce HTML Code");
        root.setAlignment(Pos.CENTER);
        showHTMLButton.setOnAction(new EventHandler<ActionEvent>() {
            @Override public void handle(ActionEvent arg0) {
                htmlCode.setText(htmlEditor.getHtmlText());
            }
        });

        root.getChildren().addAll(htmlEditor, showHTMLButton, scrollPane);
        scene.setRoot(root);

        stage.setScene(scene);
        stage.show();
    }

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

The `getHTMLText` method called on the `HTMLEditor` object derives the edited content and presents it as an HTML string. This information is passed to the text area, so that you can observe, copy, and paste the produced HTML code. Figure 19–6 shows an HTML code of the text is being edited in the HTMLEditor sample.

Figure 19–6 Obtaining the HTML Content



Similarly, you can obtain HTML code and save it in the file or send it to the `WebView` object to synchronize content in the editor and the embedded browser. See how this task is implemented in Example 19–6.

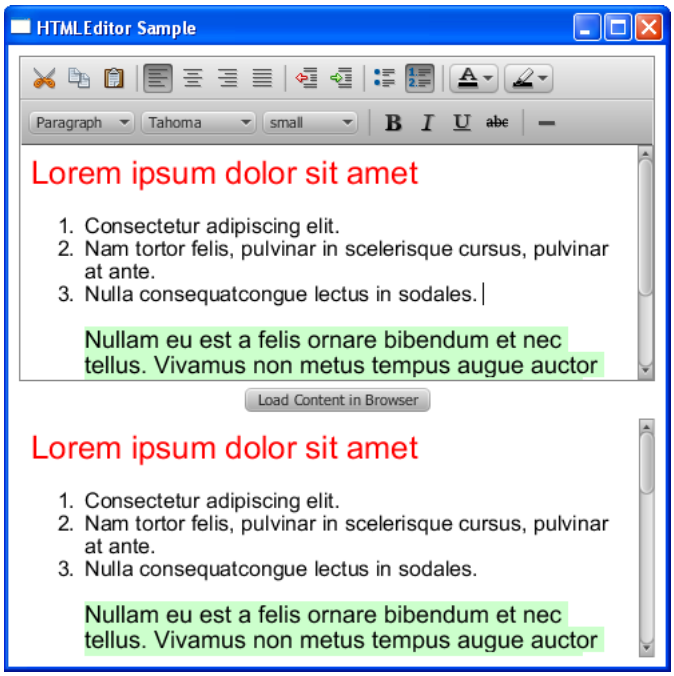
Example 19–6 Rendering Edited HTML Content in a Browser

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.VBox;
import javafx.scene.web.HTMLEditor;
import javafx.scene.web.WebEngine;
import javafx.scene.web.WebView;
import javafx.stage.Stage;

public class HTMLEditorSample extends Application {
    private final String INITIAL_TEXT = "Lorem ipsum dolor sit "
            + "amet, consectetur adipiscing elit. Nam tortor felis, pulvinar "
            + "in scelerisque cursus, pulvinar at ante. Nulla consequat"
            + "congue lectus in sodales. Nullam eu est a felis ornare "
            + "bibendum et nec tellus. Vivamus non metus tempus augue auctor "
            + "ornare. Duis pulvinar justo ac purus adipiscing pulvinar. "
            + "Integer congue faucibus dapibus. Integer id nisl ut elit "
            + "aliquam sagittis gravida eu dolor. Etiam sit amet ipsum "
            + "sem.";

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

```
        stage.setTitle("HTMLEditor Sample");
        stage.setWidth(500);
        stage.setHeight(500);
        Scene scene = new Scene(new Group());

        VBox root = new VBox();
        root.setPadding(new Insets(8, 8, 8, 8));
        root.setSpacing(5);
        root.setAlignment(Pos.BOTTOM_LEFT);

        final HTMLEditor htmlEditor = new HTMLEditor();
        htmlEditor.setPrefHeight(245);
        htmlEditor.setHtmlText(INITIAL_TEXT);

        final WebView browser = new WebView();
        final WebEngine webEngine = browser.getEngine();

        ScrollPane scrollPane = new ScrollPane();
        scrollPane.getStyleClass().add("noborder-scroll-pane");
        scrollPane.setStyle("-fx-background-color: white");
        scrollPane.setContent(browser);
        scrollPane.setFitToWidth(true);
        scrollPane.setPrefHeight(180);

        Button showHTMLButton = new Button("Load Content in Browser");
        root.setAlignment(Pos.CENTER);
        showHTMLButton.setOnAction(new EventHandler<ActionEvent>() {
            @Override public void handle(ActionEvent arg0) {
                webEngine.loadContent(htmlEditor.getHtmlText());
            }
        });

        root.getChildren().addAll(htmlEditor, showHTMLButton, scrollPane);
        scene.setRoot(root);

        stage.setScene(scene);
        stage.show();
    }

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

HTML code received from the `htmlEditor` component is loaded in the `WebEngine` object that specifies the content for the embedded browser. Each time a user clicks the Load Content in Browser button, the edited content is updated in the browser. Figure 19–7 demonstrates Example 19–6 in action.

Figure 19–7 Loading Content in a Browser



You can use the `Text` component to add non-editing text content to your UI. See Using Text and Text Effects in JavaFX for more information about the `Text` component.

Related API Documentation

- `HTMLEditor`
- `WebView`
- `WebEngine`
- `Label`
- `Button`
- `TextField`
- `ChoiceBox`
- `ScrollPane`

20

Tooltip

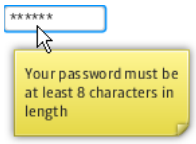
In this chapter, you learn about the tooltip, the control that can be set for any UI control to appear when that control is hovered over by the mouse cursor.

The `Tooltip` class represents a common UI component that is typically used to display additional information about the UI control. The tooltip can be set on any control by calling the `setTooltip` method.

The tooltip has two different states: activated and showing. When the tooltip is activated, the mouse moves over a control. When the tooltip is in the showing state, it actually appears. A shown tooltip is also activated. There is usually some delay between when the Tooltip becomes activated and when it is actually shown.

A password field with a tooltip is shown in Figure 20–1.

Figure 20–1 Tooltip Added to a Password Field



Creating a Tooltip

Study the code fragment in Example 20–1 that creates the password field with a tooltip in the JavaFX application shown in the preceding figure.

Example 20–1 Adding a Tooltip to the Password Field

```
final PasswordField pf = new PasswordField();
final Tooltip tooltip = new Tooltip();
tooltip.setText(
    "\nYour password must be\n" +
    "at least 8 characters in length\n"  +
);
pf.setTooltip(tooltip);
```

Each UI control from the `javafx.scene.control` package has the `setTooltip` method to add a tooltip. You can define a text caption within a `Tooltip` constructor or by using the `setText` method.

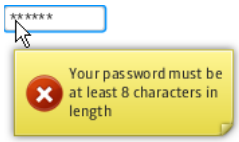
Because the `Tooltip` class is an extension of the `Labeled` class, you can add not only a text caption, but a graphical icon as well. The code fragment in Example 20–2 adds an icon to the tooltip for the password field.

Example 20–2 Adding an Icon to a Tooltip

```
Image image = new Image(
    getClass().getResourceAsStream("warn.png")
);
tooltip.setGraphic(new ImageView(image));
```

After you add this code fragment to the application, and the code is compiled, and run, the tooltip shown in Figure 20–2 appears.

Figure 20–2 Tooltip with an Icon

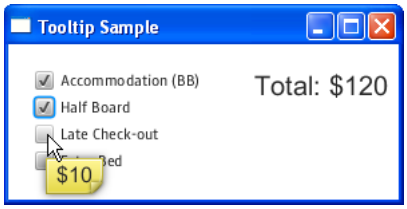


A tooltip can contain not only additional or auxiliary information, but it can also present data.

Presenting Application Data in Tooltips

The application in Figure 20–3 uses information presented in the tooltips to calculate the total cost of the hotel stay

Figure 20–3 Calculating Hotel Rates



Each checkbox is accompanied by a tooltip. Each tooltip displays the rate for a particular booking option. If a user selects a checkbox, the corresponding value is added to the total. If a selected checkbox is deselected, the corresponding value is deducted from the total.

Review the source code of the application shown in Example 20–3.

Example 20–3 Using Tooltips to Calculate Hotel Rates

```
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.geometry.Insets;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.CheckBox;
import javafx.scene.control.Label;
import javafx.scene.control.Tooltip;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
```

```
import javafx.stage.Stage;


public class Main extends Application {

    final static String[] rooms = new String[]{
        "Accommodation (BB)",
        "Half Board",
        "Late Check-out",
        "Extra Bed"
    };
    final static Integer[] rates = new Integer[]{
        100, 20, 10, 30
    };
    final CheckBox[] cbs = new CheckBox[rooms.length];
    final Label total = new Label("Total: $0");
    Integer sum = 0;

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

    @Override
    public void start(Stage stage) {
        Scene scene = new Scene(new Group());
        stage.setTitle("Tooltip Sample");
        stage.setWidth(300);
        stage.setHeight(150);

        total.setFont(new Font("Arial", 20));

        for (int i = 0; i < rooms.length; i++) {
            final CheckBox cb = cbs[i] = new CheckBox(rooms[i]);
            final Integer rate = rates[i];
            final Tooltip tooltip = new Tooltip("$" + rates[i].toString());
            tooltip.setFont(new Font("Arial", 16));
            cb.setTooltip(tooltip);
    cb.selectedProperty().addListener(new ChangeListener<Boolean>() {
                public void changed(ObservableValue<? extends Boolean> ov,
                    Boolean old_val, Boolean new_val) {
                    if (cb.isSelected()) {
                        sum = sum + rate;
                    } else {
                        sum = sum - rate;
                    }
                    total.setText("Total: $" + sum.toString());
                }
            });
        }

        VBox vbox = new VBox();
        vbox.getChildren().addAll(cbs);
        vbox.setSpacing(5);
        HBox root = new HBox();
        root.getChildren().add(vbox);
        root.getChildren().add(total);
        root.setSpacing(40);
        root.setPadding(new Insets(20, 10, 10, 20));

        ((Group) scene.getRoot()).getChildren().add(root);
```

```
        stage.setScene(scene);
        stage.show();
    }
}
```

The code line in Example 20–4 was used in Example 20–3 to create a tooltip and assign a text caption to it. The `Integer` value of the option price was converted into a `String` value.

Example 20–4 Setting the Value for a Tooltip

```
final Tooltip tooltip = new Tooltip("$" + rates[i].toString())
```

You can alter visual appearance of a tooltip by applying CSS.

Related API Documentation

- `Tooltip`
- `Labeled`

21

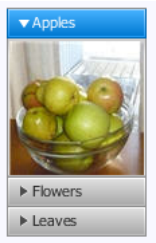
Titled Pane and Accordion

This chapter explains how to use a combination of the accordion and title panes in your JavaFX applications.

A titled pane is a panel with a title. It can be opened and closed, and it can encapsulate any `Node` such as UI controls or images, and groups of elements added to a layout container.

Titled panes can be grouped by using the accordion control, which enables you to create multiple panes and display them one at a time. Figure 21–1 shows an accordion control that combines three titled panes.

Figure 21–1 Accordion with Three Titled Panes



Use the `Accordion` and `TitledPane` classes in the JavaFX SDK API to implement these controls in your applications.

Creating Titled Panes

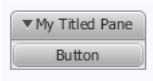
To create a `TitledPane` control define a title and some content. You can do this by using the two-parameter constructor of the `TitledPane` class, or by applying the `setText` and `setContent` methods. Both ways are shown in Example 21–1.

Example 21–1 Declaring a TitledPane Object

```
//using a two-parameter constructor
TitledPane tp = new TitledPane("My Titled Pane", new Button("Button"));
//applying methods
TitledPane tp = new TitledPane();
tp.setText("My Titled Pane");
tp.setContent(new Button("Button"));
```

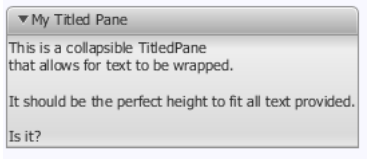
Compiling and running an application with either of the code fragments produces the control shown in Figure 21–2.

Figure 21–2 Titled Pane with a Button



The titled pane is resized to accommodate the preferred size of its content. You can add multi-line text and evaluate the result as shown in Figure 21–3.

Figure 21–3 Titled Pane with Some Text



Do not explicitly set the minimal, maximum, or preferred height of a titled pane, as this may lead to unexpected behavior when the titled pane is opened or closed.

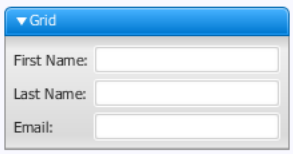
The code fragment shown in Example 21–2 adds several controls to the titled pane by putting them into the `GridPane` layout container.

Example 21–2 Titled Pane with the GridPane Layout Container

```
TitledPane gridTitlePane = new TitledPane();
GridPane grid = new GridPane();
grid.setVgap(4);
grid.setPadding(new Insets(5, 5, 5, 5));
grid.add(new Label("First Name: "), 0, 0);
grid.add(new TextField(), 1, 0);
grid.add(new Label("Last Name: "), 0, 1);
grid.add(new TextField(), 1, 1);
grid.add(new Label("Email: "), 0, 2);
grid.add(new TextField(), 1, 2);
gridTitlePane.setText("Grid");
gridTitlePane.setContent(grid);
```

When you run and compile an application with this code fragment, the output shown in Figure 21–4 appears.

Figure 21–4 Titled Pane that Contains Several Controls



You can define the way a titled pane opens and closes. By default, all titled panes are collapsible and their movements are animated. If your application prohibits closing a titled pane, use the `setCollapsible` method with the `false` value. You can also disable animated opening by applying the `setAnimated` method with the `false` value. The code fragment shown in Example 21–3 implements these tasks.

Example 21–3 Adjusting the Style of a Titled Pane

```
TitledPane tp = new TitledPane();
//prohibit closing
tp.setCollapsible(false);
//prohibit animating
tp.setAnimated(false);
```

Adding Titled Panes to an Accordion

In your applications, you can use titled panes as standalone elements, or you can combine them in a group by using the `Accordion` control. Do not explicitly set the minimal, maximum, or preferred height of an accordion, as this may lead to unexpected behavior when one of its titled pane is opened.

Adding several titled panes to an accordion is similar to adding toggle buttons to a toggle group: only one titled pane can be opened in an accordion at a time. Example 21–4 creates three titled panes and adds them to an accordion.

Example 21–4 Accordion and Three Titled Panes

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Accordion;
import javafx.scene.control.TitledPane;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class TitledPaneSample extends Application {
    final String[] imageNames = new String[]{"Apples", "Flowers", "Leaves"};
    final Image[] images = new Image[imageNames.length];
    final ImageView[] pics = new ImageView[imageNames.length];
    final TitledPane[] tps = new TitledPane[imageNames.length];

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

    @Override public void start(Stage stage) {
        stage.setTitle("TitledPane");
        Scene scene = new Scene(new Group(), 80, 180);
        scene.setFill(Color.GHOSTWHITE);

        final Accordion accordion = new Accordion ();

        for (int i = 0; i < imageNames.length; i++) {
            images[i] = new
                Image(getClass().getResourceAsStream(imageNames[i] + ".jpg"));
            pics[i] = new ImageView(images[i]);
            tps[i] = new TitledPane(imageNames[i],pics[i]);
        }
        accordion.getPanes().addAll(tps);
        accordion.setExpandedPane(tps[0]);

        Group root = (Group)scene.getRoot();
```

```
        root.getChildren().add(accordion);
        stage.setScene(scene);
        stage.show();
    }
}
```

Three titled panes are created within the loop. Content for each titled pane is defined as an `ImageView` object. The titled panes are added to the accordion by using the `getPanes` and `addAll` methods. You can use the `add` method instead of the `addAll` method to add a single titled pane.

By default, all the titled panes are closed when the application starts. The `setExpandedPane` method in Example 21–4 specifies that the titled pane with the Apples picture will be opened when you run the sample, as shown in Figure 21–5.

Figure 21–5 Accordion with Three Titled Panes

Processing Events for an Accordion with Titled Panes

You can use titled panes and accordions to present different data in your applications. Example 21–5 creates a standalone titled pane with the `GridPane` layout container and three titled panes combined by using the accordion. The standalone titled pane contains UI elements of an email client. The accordion enables the selection of an image to appear in the Attachment field of the Grid titled pane.

Example 21–5 Implementing ChangeListener for an Accordion

```
import javafx.application.Application;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.geometry.Insets;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.Accordion;
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.scene.paint.Color;
import javafx.stage.Stage;

public class TitledPaneSample extends Application {
    final String[] imageNames = new String[]{"Apples", "Flowers", "Leaves"};
    final Image[] images = new Image[imageNames.length];
    final ImageView[] pics = new ImageView[imageNames.length];
```

```
    final TitledPane[] tps = new TitledPane[imageNames.length];
    final Label label = new Label("N/A");

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

    @Override public void start(Stage stage) {
        stage.setTitle("TitledPane");
        Scene scene = new Scene(new Group(), 800, 250);
        scene.setFill(Color.GHOSTWHITE);

        // --- GridPane container
        TitledPane gridTitlePane = new TitledPane();
        GridPane grid = new GridPane();
        grid.setVgap(4);
        grid.setPadding(new Insets(5, 5, 5, 5));
        grid.add(new Label("To: "), 0, 0);
        grid.add(new TextField(), 1, 0);
        grid.add(new Label("Cc: "), 0, 1);
        grid.add(new TextField(), 1, 1);
        grid.add(new Label("Subject: "), 0, 2);
        grid.add(new TextField(), 1, 2);
        grid.add(new Label("Attachment: "), 0, 3);
        grid.add(label,1, 3);
        gridTitlePane.setText("Grid");
        gridTitlePane.setContent(grid);

        // --- Accordion
        final Accordion accordion = new Accordion ();
        for (int i = 0; i < imageNames.length; i++) {
            images[i] = new
                Image(getClass().getResourceAsStream(imageNames[i] + ".jpg"));
            pics[i] = new ImageView(images[i]);
            tps[i] = new TitledPane(imageNames[i],pics[i]);
        }
        accordion.getPanes().addAll(tps);
        accordion.expandedPaneProperty().addListener(new
            ChangeListener<TitledPane>() {
                public void changed(ObservableValue<? extends TitledPane> ov,
                    TitledPane old_val, TitledPane new_val) {
                        if (new_val != null) {
                            label.setText(accordion.getExpandedPane().getText() +
                                ".jpg");
                        }
                }
        });

        HBox hbox = new HBox(10);
        hbox.setPadding(new Insets(20, 0, 0, 20));
        hbox.getChildren().setAll(gridTitlePane, accordion);

        Group root = (Group)scene.getRoot();
        root.getChildren().add(hbox);
        stage.setScene(scene);
        stage.show();
    }
}
```

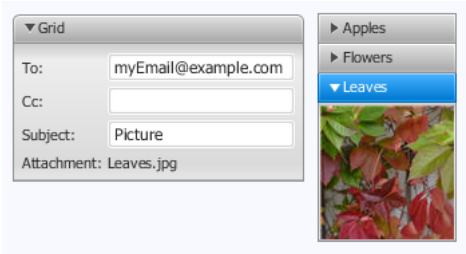
When a user opens a titled pane in the accordion, the `expandedPaneProperty` of the accordion changes. The `ChangeListener` object is notified about this change, and the expanded titled pane in the accordion is used to construct a file name of the attachment. This file name is set as text of the corresponding `Label` object.

Figure 21–6 shows how the application looks after its start. The Attachment label contains "N/A," because none of the titled panes are selected.

Figure 21–6 Initial View of the TitledPaneSample Application

If you expand the Leaves titled pane, the Attachment label will contain "Leaves.jpg," as shown in Figure 21–7.

Figure 21–7 TitledPaneSample Application with the Leaves Titled Pane Expanded



Because the `TitledPane` and `Accordion` classes are both extensions of the `Node` class, you can apply visual effects or transformations to them. You can also change the appearance of the controls by applying CSS styles.

Related API Documentation

- `TitledPane`
- `Accordion`
- `Label`
- `GridPane`
- `TextField`

22

Menu

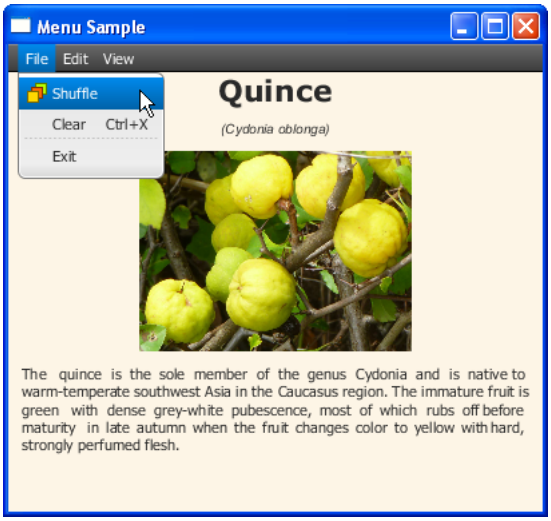
This chapter explains how to create menus and menu bars, add menu items, group the menus into categories, create submenus, and set context menus.

You can use the following classes of the JavaFX API to build menus in your JavaFX application.

- MenuBar
- MenuItem
  - Menu
  - CheckMenuItem
  - RadioMenuItem
  - Menu
  - CustomMenuItem
    - SeparatorMenuItem
- ContextMenu

Figure 22–1 shows a screen capture of an application with a typical menu bar.

Figure 22–1 Application with a Menu Bar and Three Menu Categories



Building Menus in JavaFX Applications

A menu is a list of actionable items that can be displayed upon a user's request. When a menu is visible, users can select one menu item at time. After a user clicks an item, the menu returns to the hidden mode. By using menus, you can save space in your application user interface (UI) by placing in menus the functionality that does not always need to be visible.

The menus in a menu bar are typically grouped into categories. The coding pattern is to declare a menu bar, define the category menus, and populate the category menus with menu items. Use the following menu item classes when building menus in your JavaFX applications:

- `MenuItem` – to create one actionable option
- `Menu` – to create a submenu
- `RadioButtonItem` – to create a mutually exclusive selection
- `CheckMenuItem` – to create an option that can be toggled between selected and unselected states

To separate menu items within one category, use the `SeparatorMenuItem` class.

The menus organized by categories in a menu bar are typically located at the top of the window, leaving the rest of the scene for crucial UI elements. If, for some reasons, you cannot allot any visual part of your UI for a menu bar, you can use context menus that the user opens with a mouse click.

Creating a Menu Bar

Although a menu bar can be placed elsewhere in the user interface, typically it is located at the top of the UI and it contains one or more menus. The menu bar automatically resizes to fit the width of the application window. By default, each menu added to the menu bar is represented by a button with the text value.

Consider an application that renders reference information about plants such as their name, binomial name, picture, and a brief description. You can create three menu categories: File, Edit, and View, and populate them with the menu items. Example 22–1 shows the source code of such an application with the menu bar added.

Example 22–1 Menu Sample Application

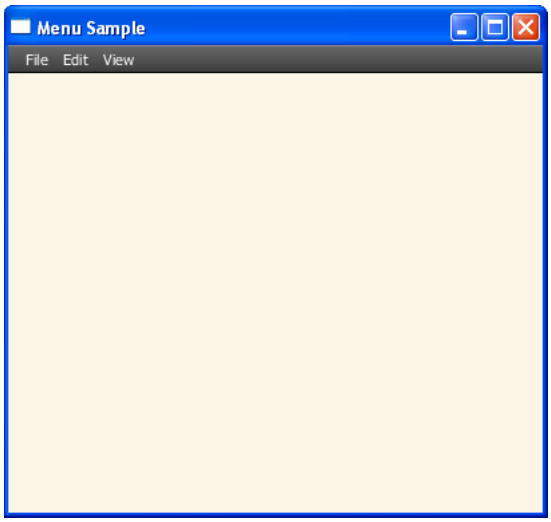
```
import java.util.AbstractMap.SimpleEntry;
import java.util.Map.Entry;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.effect.DropShadow;
import javafx.scene.effect.Effect;
import javafx.scene.effect.Glow;
import javafx.scene.effect.SepiaTone;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class MenuSample extends Application {
```

```
final PageData[] pages = new PageData[] {
    new PageData("Apple",
        "The apple is the pomaceous fruit of the apple tree, species Malus "
        + "domestica in the rose family (Rosaceae). It is one of the most "
        + "widely cultivated tree fruits, and the most widely known of "
        + "the many members of genus Malus that are used by humans. "
        + "The tree originated in Western Asia, where its wild ancestor, "
        + "the Alma, is still found today.",
        "Malus domestica"),
    new PageData("Hawthorn",
        "The hawthorn is a large genus of shrubs and trees in the rose "
        + "family, Rosaceae, native to temperate regions of the Northern "
        + "Hemisphere in Europe, Asia and North America. "
        + The name hawthorn was "
        + "originally applied to the species native to northern Europe, "
        + "especially the Common Hawthorn C. monogyna, and the unmodified "
        + "name is often so used in Britain and Ireland.",
        "Crataegus monogyna"),
    new PageData("Ivy",
        "The ivy is a flowering plant in the grape family (Vitaceae) native to
        + " eastern Asia in Japan, Korea, and northern and eastern China. "
        + "It is a deciduous woody vine growing to 30 m tall or more given "
        + "suitable support,  attaching itself by means of numerous small "
        + "branched tendrils tipped with sticky disks.",
        "Parthenocissus tricuspidata"),
    new PageData("Quince",
        "The quince is the sole member of the genus Cydonia and is native to "
        + "warm-temperate southwest Asia in the Caucasus region. The "
        + "immature fruit is green with dense grey-white pubescence, most "
        + "of which rubs off before maturity in late autumn when the fruit "
        + "changes color to yellow with hard, strongly perfumed flesh.",
        "Cydonia oblonga")
};

final String[] viewOptions = new String[] {
    "Title",
    "Binomial name",
    "Picture",
    "Description"
};

final Entry<String, Effect>[] effects = new Entry[] {
    new SimpleEntry<String, Effect>("Sepia Tone", new SepiaTone()),
    new SimpleEntry<String, Effect>("Glow", new Glow()),
    new SimpleEntry<String, Effect>("Shadow", new DropShadow())
};

final ImageView pic = new ImageView();
final Label name = new Label();
final Label binName = new Label();
final Label description = new Label();

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

@Override
public void start(Stage stage) {
    stage.setTitle("Menu Sample");
```

```
        Scene scene = new Scene(new VBox(), 400, 350);
        scene.setFill(Color.OLDLACE);

        MenuBar menuBar = new MenuBar();

        // --- Menu File
        Menu menuFile = new Menu("File");

        // --- Menu Edit
        Menu menuEdit = new Menu("Edit");

        // --- Menu View
        Menu menuView = new Menu("View");

        menuBar.getMenus().addAll(menuFile, menuEdit, menuView);


        ((VBox) scene.getRoot()).getChildren().addAll(menuBar);

        stage.setScene(scene);
        stage.show();
    }


    private class PageData {
        public String name;
        public String description;
        public String binNames;
        public Image image;
        public PageData(String name, String description, String binNames) {
            this.name = name;
            this.description = description;
            this.binNames = binNames;
            image = new Image(getClass().getResourceAsStream(name + ".jpg"));
        }
    }
}
```

Unlike other UI Controls, the `Menu` class and other extensions of the `MenuItem` class do not extend the `Node` class. They cannot be added directly to the application scene and remain invisible until added to the menu bar through the `getMenus` method.

Figure 22–2 Menu Bar is Added to the Application



You can navigate through the menus by using the arrow keys of the keyboard. However, when you select a menu, no action is performed, because the behavior for the menus is not defined yet.

Adding Menu Items

Set the functionality for the File menu by adding the following items:

- Shuffle – to load reference information about plants
- Clear – to remove the reference information and clear the scene
- Separator – to detach menu items
- Exit – to exit the application

Bold lines in Example 22–2 create a Shuffle menu by using the `MenuItem` class and add graphical components to the application scene. The `MenuItem` class enables creating an actionable item with text and graphics. The action performed on a user click is defined by the `setOnAction` method, similar to the `Button` class.

Example 22–2 Adding the Shuffle Menu Item with Graphics

```
import java.util.AbstractMap.SimpleEntry;
import java.util.Map.Entry;
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.effect.DropShadow;
import javafx.scene.effect.Effect;
import javafx.scene.effect.Glow;
import javafx.scene.effect.SepiaTone;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
```

```
import javafx.scene.text.Font;
import javafx.scene.text.TextAlignment;
import javafx.stage.Stage;

public class MenuSample extends Application {

    final PageData[] pages = new PageData[] {
        new PageData("Apple",
            "The apple is the pomaceous fruit of the apple tree, species Malus "
            +"domestica in the rose family (Rosaceae). It is one of the most "
            +"widely cultivated tree fruits, and the most widely known of "
            +"the many members of genus Malus that are used by humans. "
            +"The tree originated in Western Asia, where its wild ancestor, "
            +"the Alma, is still found today.",
            "Malus domestica"),
        new PageData("Hawthorn",
            "The hawthorn is a large genus of shrubs and trees in the rose "
            + "family, Rosaceae, native to temperate regions of the Northern "
            + "Hemisphere in Europe, Asia and North America. "
            + "The name hawthorn was "
            + "originally applied to the species native to northern Europe, "
            + "especially the Common Hawthorn C. monogyna, and the unmodified "
            + "name is often so used in Britain and Ireland.",
            "Crataegus monogyna"),
        new PageData("Ivy",
            "The ivy is a flowering plant in the grape family (Vitaceae) native"
            +" to eastern Asia in Japan, Korea, and northern and eastern China."
            +" It is a deciduous woody vine growing to 30 m tall or more given "
            +"suitable support,  attaching itself by means of numerous small "
            +"branched tendrils tipped with sticky disks.",
            "Parthenocissus tricuspidata"),
        new PageData("Quince",
            "The quince is the sole member of the genus Cydonia and is native"
            +" to warm-temperate southwest Asia in the Caucasus region. The "
            +"immature fruit is green with dense grey-white pubescence, most "
            +"of which rubs off before maturity in late autumn when the fruit "
            +"changes color to yellow with hard, strongly perfumed flesh.",
            "Cydonia oblonga")
    };

    final String[] viewOptions = new String[] {
        "Title",
        "Binomial name",
        "Picture",
        "Description"
    };

    final Entry<String, Effect>[] effects = new Entry[] {
        new SimpleEntry<String, Effect>("Sepia Tone", new SepiaTone()),
        new SimpleEntry<String, Effect>("Glow", new Glow()),
        new SimpleEntry<String, Effect>("Shadow", new DropShadow())
    };

    final ImageView pic = new ImageView();
    final Label name = new Label();
    final Label binName = new Label();
    final Label description = new Label();
    private int currentIndex = -1;

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

```
        launch(args);
    }

    @Override
    public void start(Stage stage) {
        stage.setTitle("Menu Sample");
        Scene scene = new Scene(new VBox(), 400, 350);
        scene.setFill(Color.OLDLACE);

        name.setFont(new Font("Verdana Bold", 22));
        binName.setFont(new Font("Arial Italic", 10));
        pic.setFitHeight(150);
        pic.setPreserveRatio(true);
        description.setWrapText(true);
        description.setTextAlignment(TextAlignment.JUSTIFY);

        shuffle();

        MenuBar menuBar = new MenuBar();

        final VBox vbox = new VBox();
        vbox.setAlignment(Pos.CENTER);
        vbox.setSpacing(10);
        vbox.setPadding(new Insets(0, 10, 0, 10));
        vbox.getChildren().addAll(name, binName, pic, description);

        // --- Menu File
        Menu menuFile = new Menu("File");
        MenuItem add = new MenuItem("Shuffle",
            new ImageView(new Image("menusample/new.png")));
        add.setOnAction(new EventHandler<ActionEvent>() {
            public void handle(ActionEvent t) {
                shuffle();
                vbox.setVisible(true);
            }
        });

        menuFile.getItems().addAll(add);

        // --- Menu Edit
        Menu menuEdit = new Menu("Edit");

        // --- Menu View
        Menu menuView = new Menu("View");

        menuBar.getMenus().addAll(menuFile, menuEdit, menuView);
        ((VBox) scene.getRoot()).getChildren().addAll(menuBar, vbox);
        stage.setScene(scene);
        stage.show();
    }

    private void shuffle() {
        int i = currentIndex;
        while (i == currentIndex) {
            i = (int) (Math.random() * pages.length);
        }
        pic.setImage(pages[i].image);
        name.setText(pages[i].name);
        binName.setText("(" + pages[i].binNames + ")");
        description.setText(pages[i].description);
```

```
            currentIndex = i;
        }


        private class PageData {
            public String name;
            public String description;
            public String binNames;
            public Image image;
            public PageData(String name, String description, String binNames) {
                this.name = name;
                this.description = description;
                this.binNames = binNames;
                image = new Image(getClass().getResourceAsStream(name + ".jpg"));
            }
        }
}
```

When a user selects the Shuffle menu item, the `shuffle` method called within `setOnAction` specifies the title, the binomial name, a picture of the plant, and its description by calculating the index of the elements in the corresponding arrays.

The Clear menu item is used to erase the application scene. You can implement this by making the `VBox` container with the GUI elements invisible as shown in Example 22–3.

Example 22–3 Creating the Clear Menu Item with Accelerator

```
MenuItem clear = new MenuItem("Clear");
    clear.setAccelerator(KeyCombination.keyCombination("Ctrl+X"));
    clear.setOnAction(new EventHandler<ActionEvent>() {
        public void handle(ActionEvent t) {
            vbox.setVisible(false);
        }
});
```

Implementation of the `MenuItem` class enables developers to set a menu accelerator, a key combination that performs the same action as the menu item. With the Clear menu, users can either select the action from the File menu category or press Control Key and X key simultaneously.

The Exit menu closes the application window. Set `System.exit(0)` as an action for this menu item as shown in Example 22–4.

Example 22–4 Creating the Exit Menu Item

```
MenuItem exit = new MenuItem("Exit");
exit.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent t) {
        System.exit(0);
    }
});
```

Use the `getItems` method shown in Example 22–5 to add the newly created menu items to the File menu. You can create a separator menu item and add it within the `getItems` method to visually detach the Exit menu item.

Example 22–5 Adding Menu Items

```
menuFile.getItems().addAll(add, clear, new SeparatorMenuItem(), exit);
```

Add Example 22–2, Example 22–3, Example 22–4, and Example 22–5 to the Menu Sample application, and then compile and run it. Select the Shuffle menu item to load reference information about different plants. Then clear the scene (Clear), and close the application (Exit). Figure 22–3 shows selection of the Clear menu item.

Figure 22–3 File Menu with Three Menu Items

With the View menu, you can hide and show elements of reference information. Implement the `createMenuItem` method and call it within the `start` method to create four `CheckMenuItem` objects. Then add newly created check menu items to the View menu to toggle visibility of the title, binomial name, picture of the plant, and its description. Example 22–6 shows two code fragments that implement these tasks.

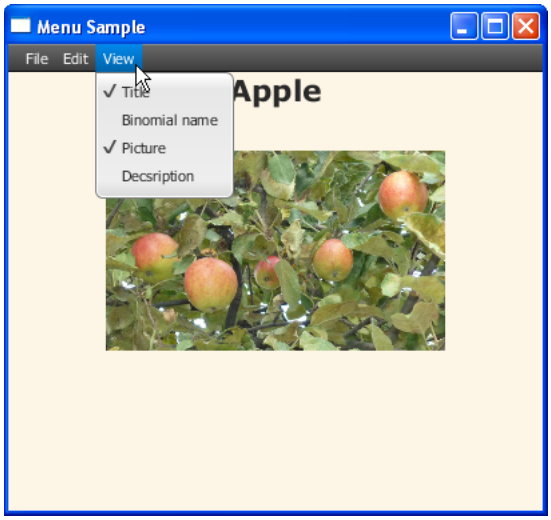
Example 22–6 Applying the CheckMenuItem Class to Create Toggle Options

```
// --- Creating four check menu items within the start method
CheckMenuItem titleView = createMenuItem ("Title", name);
CheckMenuItem binNameView = createMenuItem ("Binomial name", binName);
CheckMenuItem picView = createMenuItem ("Picture", pic);
CheckMenuItem descriptionView = createMenuItem ("Description", description);
menuView.getItems().addAll(titleView, binNameView, picView, descriptionView);

...

// The createMenuItem method
private static CheckMenuItem createMenuItem (String title, final Node node){
    CheckMenuItem cmi = new CheckMenuItem(title);
    cmi.setSelected(true);
    cmi.selectedProperty().addListener(new ChangeListener<Boolean>() {
        public void changed(ObservableValue ov,
        Boolean old_val, Boolean new_val) {
            node.setVisible(new_val);
        }
    });
    return cmi;
}
```

The `CheckMenuItem` class is an extension of the `MenuItem` class. It can be toggled between selected and deselected states. When selected, a check menu item shows a check mark.

Example 22–6 creates four `CheckMenuItem` objects and processes the changing of their `selectedProperty` property. When, for example, a user deselects the `picView` item, the `setVisible` method receives the `false` value, the picture of the plant becomes invisible. When you add this code fragment to the application, compile, and run the application, you can experiment with selecting and deselecting the menu items. Figure 22–4 shows the application in the moment when the title and picture of the plant are shown, but its binomial name and description are hidden.

Figure 22–4 Using Check Menu Items



Creating Submenus

For the Edit menu, define two menu items: Picture Effect and No Effects. The Picture Effect menu item is designed as a submenu with three items to set one of the three available visual effects. The No Effects menu item removes the selected effect and restores the initial state of the image.

Use the `RadioMenuItem` class to create the items of the submenu. Add the radio menu buttons to a toggle group to make the selection mutually exclusive. Example 22–7 implements these tasks.

Example 22–7 Creating a Submenu with Radio Menu Items

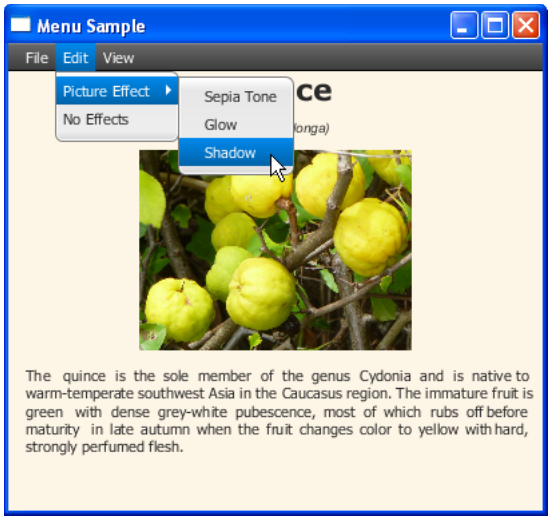
```
//Picture Effect menu
Menu menuEffect = new Menu("Picture Effect");
final ToggleGroup groupEffect = new ToggleGroup();
for (Entry<String, Effect> effect : effects) {
    RadioMenuItem itemEffect = new RadioMenuItem(effect.getKey());
    itemEffect.setUserData(effect.getValue());
    itemEffect.setToggleGroup(groupEffect);
    menuEffect.getItems().add(itemEffect);
}
//No Effects menu
final MenuItem noEffects = new MenuItem("No Effects");
```

```
 noEffects.setOnAction(new EventHandler<ActionEvent>() {
     public void handle(ActionEvent t) {
         pic.setEffect(null);
         groupEffect.getSelectedToggle().setSelected(false);
     }
});

//Processing menu item selection
groupEffect.selectedToggleProperty().addListener(new ChangeListener<Toggle>() {
    public void changed(ObservableValue<? extends Toggle> ov,
        Toggle old_toggle, Toggle new_toggle) {
            if (groupEffect.getSelectedToggle() != null) {
                Effect effect =
                    (Effect) groupEffect.getSelectedToggle().getUserData();
                pic.setEffect(effect);
            }
        }
 });
//Adding items to the Edit menu
menuEdit.getItems().addAll(menuEffect, noEffects);
```

The `setUserData` method defines a visual effect for a particular radio menu item. When one of the items in the toggle group is selected, the corresponding effect is applied to the picture. When the No Effects menu item is selected, the `setEffect` method specifies the `null` value and no effects are applied to the picture.

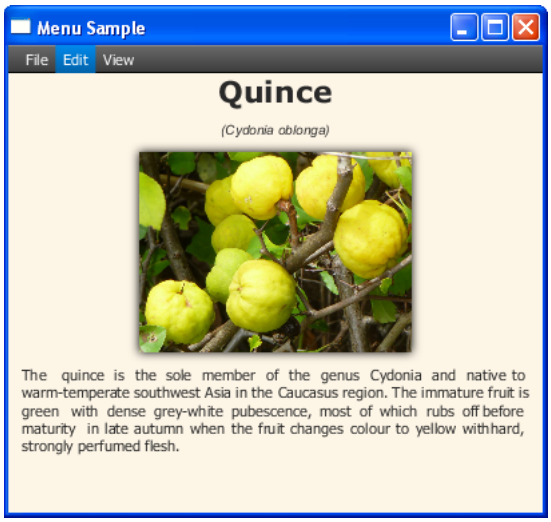
Figure 22–5 captures a moment when a user is selecting a Shadow menu item.

Figure 22–5 Submenu with Three Radio Menu Items



When the `DropShadow` effect is applied to the picture, it looks as shown in Figure 22–6.

Figure 22–6 Picture of Quince with a DropShadow Effect Applied



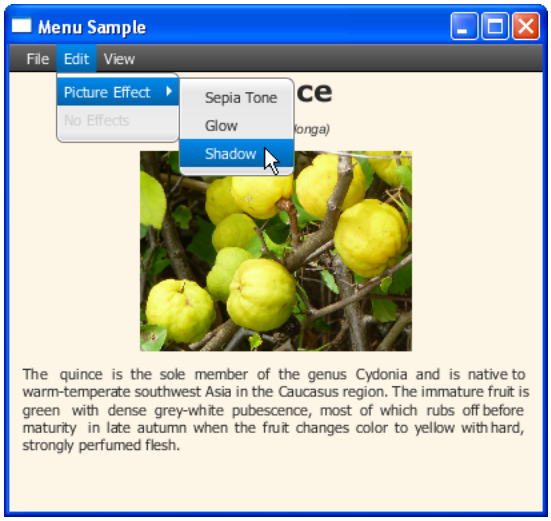
You can use the `setDisable` method of the `MenuItem` class to disable the No Effects menu when none of the effects are selected in the Picture Effect submenu. Modify Example 22–7 as shown in Example 22–8.

Example 22–8 Disabling a Menu Item

```
Menu menuEffect = new Menu("Picture Effect");
final ToggleGroup groupEffect = new ToggleGroup();
for (Entry<String, Effect> effect : effects) {
     RadioMenuItem itemEffect = new RadioMenuItem(effect.getKey());
     itemEffect.setUserData(effect.getValue());
     itemEffect.setToggleGroup(groupEffect);
     menuEffect.getItems().add(itemEffect);
}
final MenuItem noEffects = new MenuItem("No Effects");
noEffects.setDisable(true);
noEffects.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent t) {
        pic.setEffect(null);
        groupEffect.getSelectedToggle().setSelected(false);
        noEffects.setDisable(true);
    }
});

groupEffect.selectedToggleProperty().addListener(new ChangeListener<Toggle>() {
    public void changed(ObservableValue<? extends Toggle> ov,
        Toggle old_toggle, Toggle new_toggle) {
            if (groupEffect.getSelectedToggle() != null) {
                Effect effect =
                    (Effect) groupEffect.getSelectedToggle().getUserData();
                pic.setEffect(effect);
                noEffects.setDisable(false);
            } else {
                noEffects.setDisable(true);
        }
    }
});
menuEdit.getItems().addAll(menuEffect, noEffects);
```

When none of the `RadioMenuItem` options are selected, the No Effect menu item is disabled as shown in Figure 22–7. When a user selects one of the visual effects, the No Effects menu item is enabled.

Figure 22–7 Effect Menu Item Is Disabled



Adding Context Menus

When you cannot allocate any space of your user interface for a required functionality, you can use a context menu. A context menu is a pop-up window that appears in response to a mouse click. A context menu can contain one or more menu items.

In the Menu Sample application, set a context menu for the picture of the plant, so that users can copy the image.

Use the `ContextMenu` class to define the context menu as shown in Example 22–9.

Example 22–9 Defining a Context Menu

```
final ContextMenu cm = new ContextMenu();
MenuItem cmItem1 = new MenuItem("Copy Image");
cmItem1.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent e) {
        Clipboard clipboard = Clipboard.getSystemClipboard();
        ClipboardContent content = new ClipboardContent();
        content.putImage(pic.getImage());
        clipboard.setContent(content);
    }
});

cm.getItems().add(cmItem1);
pic.addEventHandler(MouseEvent.MOUSE_CLICKED,
    new EventHandler<MouseEvent>() {
        @Override public void handle(MouseEvent e) {
            if (e.getButton() == MouseButton.SECONDARY)
                cm.show(pic, e.getScreenX(), e.getScreenY());
        }
});
```

When a user right clicks the `ImageView` object, the `show` method is called for the context menu to enable its showing.

The `setOnAction` method defined for the Copy Image item of the context menu creates a `Clipboard` object and adds the image as its content. Figure 22–8 captures a moment when a user is selecting the Copy Image context menu item.

Figure 22–8 Using the Context Menu

You can try to copy the image and paste it into in a graphical editor.

For further enhancements, you can add more menu items to the context menu and specify different actions. You can also create a custom menu by using the `CustomMenuItem` class. With this class you can embed an arbitrary node within a menu and specify, for example, a button or a slider as a menu item.

Related API Documentation

- `Menu`
- `MenuItem`
- `RadioMenuItem`
- `CheckMenuItem`
- `ContextMenu`
- `SeparatorMenuItem`
- `CustomMenuItem`

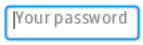
23

Password Field

In this chapter, you learn about yet another type of the text control, the password field.

The `PasswordField` class implements a specialized text field. The characters typed by a user are hidden by displaying an echo string. Figure 23–1 shows a password field with a prompt message in it.

Figure 23–1 Password Field with a Prompt Message



Creating a Password Field

An entry-level task is to create a password field by using the code in Example 23–1.

Example 23–1 Creating a Password Field

```
PasswordField passwordField = new PasswordField();
passwordField.setPromptText("Your password");
```

For your user interface, you can accompany the password field with a prompt message or you can add a notifying label. As with the `TextField` class, the `PasswordField` class provides the `setText` method to render a text string in the control when the application is started. However, the string specified in the `setText` method is hidden by the echo characters in the password field. By default, the echo character is an asterisk. Figure 23–2 shows the password field with the predefined text in it.

Figure 23–2 Password Field with the Set Text

The value typed in the password field can be obtained through the `getText` method. You can process this value in your application and set the authentication logic as appropriate.

Evaluating the Password

Take a moment to review in Example 23–2 the implementation of a password field that you can apply in your user interface.

Example 23–2 Implementing the Authentication Logic

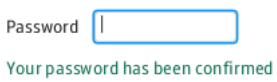
```
final Label message = new Label("");

VBox vb = new VBox();
vb.setPadding(new Insets(10, 0, 0, 10));
vb.setSpacing(10);
HBox hb = new HBox();
hb.setSpacing(10);
hb.setAlignment(Pos.CENTER_LEFT);

Label label = new Label("Password");
final PasswordField pb = new PasswordField();

pb.setOnAction(new EventHandler<ActionEvent>() {
    @Override public void handle(ActionEvent e) {
        if (!pb.getText().equals("T2f$Ay!")) {
            message.setText("Your password is incorrect!");
            message.setTextFill(Color.rgb(210, 39, 30));
        } else {
            message.setText("Your password has been confirmed");
            message.setTextFill(Color.rgb(21, 117, 84));
        }
        pb.clear();
    }
});

hb.getChildren().addAll(label, pb);
vb.getChildren().addAll(hb, message);
```

The authentication logic of the password field is defined by using the `setOnAction` method. This method is called when a password is committed and it creates a new `EventHandler` object to process the typed value. If the typed value is different from the required password, the corresponding message appears in red as shown in Figure 23–3.

Figure 23–3 Password is Incorrect

If the typed value satisfies the predefined criteria, the confirmation message appears as shown in Figure 23–4.

Figure 23–4 Password is Correct



For security reasons, it is good practice to clear the password field after the value is typed. In Example 23–2, an empty string is set for the passwordField after the authentication is performed.

Related API Documentation

- `PasswordField`
- `TextInputControl`

24

Color Picker

This chapter describes the `ColorPicker` control, provides its design overview, and explains how to use it in your JavaFX applications.

The color picker control in the JavaFX SDK is a typical user interface component that enables users to select a particular color from the available range, or set an additional color by specifying an RGB or HSB combination.

Design Overview

The `ColorPicker` control consists of the color chooser, color palette, and custom color dialog window. Figure 24–1 shows these elements.

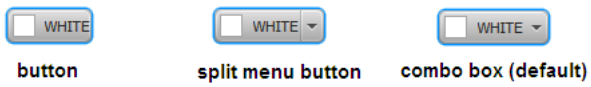
Figure 24–1 Elements of a Color Picker Control

Color Chooser

The color chooser element of the color picker is the combo box with the enabled color indicator and the corresponding label shown at the top of Figure 24–1. The color indicator presents the currently selected color.

The implementation of the color picker control allows three looks of the color chooser element: button, split-menu-button, and combo box shown in Figure 24–2.

Figure 24–2 Views of a Color Chooser



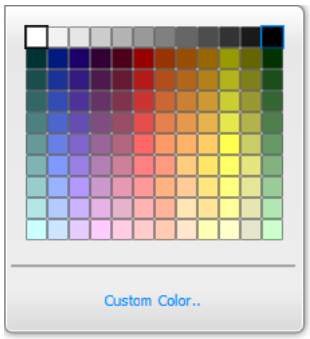
The button look provides a typical button control with the color indicator and the label. In the split-menu-button look, the button part of the control is separated from the drop-down menu. The combo-box look is the default appearance of the color chooser. It also has a drop-down menu but it is not separated from the button part.

To apply one of the looks, use the corresponding CSS classes. For more information about how to alter the appearance of a color picker, see Changing the Appearance of a Color Picker.

Color Palette

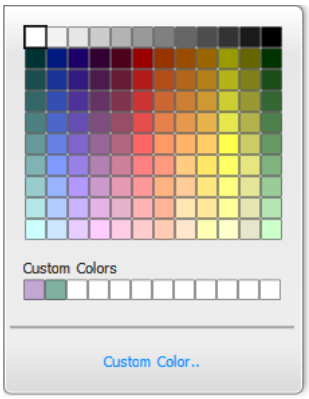
The color palette contains the predefined set of colors and the Custom Color link that points to the Custom Color dialog window. The initial appearance of the color palette is shown in Figure 24–3.

Figure 24–3 Initial Appearance of the Color Palette



If a custom color is already defined, this color is displayed in the Custom Color area in the color palette, as shown in Figure 24–4.

Figure 24–4 Color Palette with the Custom Color Area



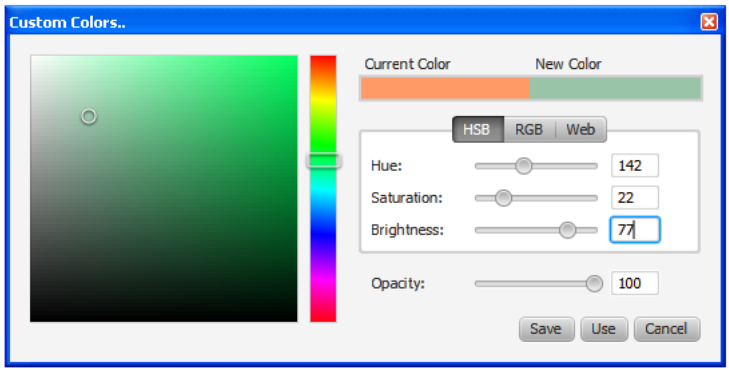
The color palette supports navigation by using the Up, Down, Left and Right keys.

The set of custom colors cannot be reloaded when the application starts again, unless saved within the application. Each color selected in the palette or in the custom color area is displayed in the color indicator of the color chooser. The color chooser label renders the corresponding hexadecimal web color value.

Custom Color Dialog Window

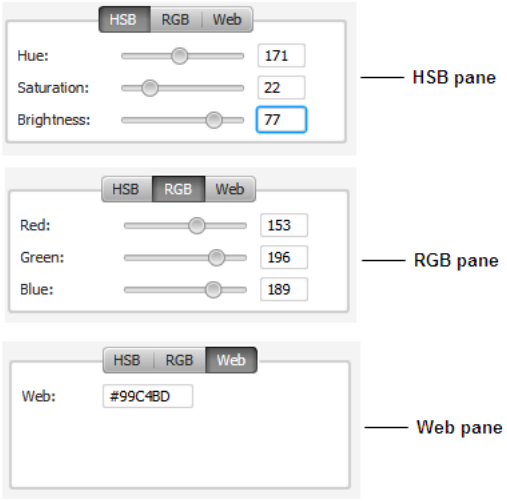
The Custom Color dialog window is a modal window that can be opened by clicking the corresponding link in the color palette. When the Custom Color window opens, it displays values of the color that is currently shown in the color indicator of the color chooser. Users can define a new color by moving the mouse cursor over the color area or over the vertical color bar, shown in Figure 24–5. Note that any time a user manipulates with the circle in the color area or with the rectangle in the color bar, the color value is automatically assigned to the corresponding property of the `ColorPicker` control.

Figure 24–5 Custom Colors Dialog Window



Another way to define a new color is to set the HSB (Hue Saturation Brightness) or RGB (Red Green Blue) values, or explicitly enter the web color value in the corresponding field. Figure 24–6 shows three panes for the custom color settings.

Figure 24–6 Color Setting Panes in the Custom Colors Dialog Window



Users can also set the transparency of the custom color by moving the Opacity slider or typing the value from 0 to 100.

When all the settings are done and the new color is specified, users can click Use to apply it, or Save to save the color to the custom color area.

Using a Color Picker

Use the `ColorPicker` class of the JavaFX SDK to build a color picker in your JavaFX application. You can add a color picker directly to the application scene, to a layout container, or to the application toolbar. Example 24–1 shows three ways to declare a `ColorPicker` object.

Example 24–1 Creating a Color Picker Control

```
//Empty contructor, the control appears with the default color, which is WHITE
ColorPicker colorPicker1 = new ColorPicker();
//Color constant set as the currently selected color
ColorPicker colorPicker2 = new ColorPicker(Color.BLUE);
//Web color value set as the currently selected color
ColorPicker colorPicker3 = new ColorPicker(Color.web("#ffcce6"));
```

Try the sample shown in Example 24–2 to evaluate the `ColorPicker` control in action.

Example 24–2 Using the ColorPicker Control to Alter the Color of the Text Component

```
import javafx.application.Application;
import javafx.event.*;
import javafx.scene.Scene;
import javafx.scene.control.ColorPicker;
import javafx.geometry.Insets;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.scene.text.*;
import javafx.stage.Stage;

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

```
        launch(args);
    }

    @Override
    public void start(Stage stage) {
        stage.setTitle("ColorPicker");
        Scene scene = new Scene(new HBox(20), 400, 100);
        HBox box = (HBox) scene.getRoot();
        box.setPadding(new Insets(5, 5, 5, 5));

        final ColorPicker colorPicker = new ColorPicker();
        colorPicker.setValue(Color.CORAL);

        final Text text = new Text("Try the color picker!");
        text.setFont(Font.font ("Verdana", 20));
        text.setFill(colorPicker.getValue());

        colorPicker.setOnAction(new EventHandler() {
            public void handle(Event t) {
                text.setFill(colorPicker.getValue());
            }
        });

        box.getChildren().addAll(colorPicker, text);

        stage.setScene(scene);
        stage.show();
    }
}
```

This example creates a color picker and defines its behavior when the color is changed. The `Color` value obtained through the `getValue` method of the `ColorPicker` class is passed to the `setFill` method of the `Text` object. This is how the selected color is applied to the "Try the color picker!" text.

When you compile and run this sample, it produces the output shown in Figure 24–7. The figure captures the moment when a newly created custom color is applied to the `Text` component.

Figure 24–7 ColorPicker and a Text Component

Similarly, you can apply the selected color to the graphical `Node`. Example 24–3 shows how to use a color picker to model a T-shirt.

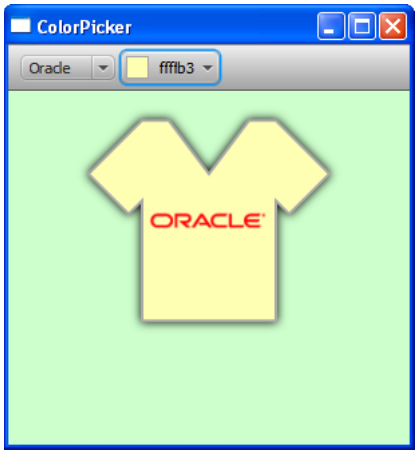
Example 24–3 Using ColorPicker to Alter the Color of a Graphical Object

```
import javafx.application.Application;
import javafx.event.Event;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.ColorPicker;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.scene.control.ComboBox;
import javafx.scene.control.ToolBar;
import javafx.scene.effect.DropShadow;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.StackPane;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.shape.SVGPath;
import javafx.stage.Stage;

public class ColorPickerSample extends Application {

    ImageView logo = new ImageView(
        new Image(getClass().getResourceAsStream("OracleLogo.png"))
    );

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

    @Override
    public void start(Stage stage) {
        stage.setTitle("ColorPicker");

        Scene scene = new Scene(new VBox(20), 300, 300);
        scene.setFill(Color.web("#ccffcc"));
        VBox box = (VBox) scene.getRoot();

        ToolBar tb = new ToolBar();
        box.getChildren().add(tb);

        final ComboBox logoSamples = new ComboBox();
        logoSamples.getItems().addAll(
                "Oracle",
                "Java",
                "JavaFX",
                "Cup");
        logoSamples.setValue("Oracle");

        logoSamples.valueProperty().addListener(new ChangeListener<String>() {
            @Override
          public void changed(ObservableValue ov, String t, String t1) {
                logo.setImage(
                    new Image(getClass().getResourceAsStream(t1+"Logo.png"))
                );
            }
        });
```

```
        final ColorPicker colorPicker = new ColorPicker();
        tb.getItems().addAll(logoSamples, colorPicker);

        StackPane stack = new StackPane();
        box.getChildren().add(stack);

        final SVGPath svg = new SVGPath();
        svg.setContent("M70,50 L90,50 L120,90 L150,50 L170,50"
            + "L210,90 L180,120 L170,110 L170,200 L70,200 L70,110 L60,120 L30,90"
            + "L70,50");
        svg.setStroke(Color.DARKGREY);
        svg.setStrokeWidth(2);
        svg.setEffect(new DropShadow());
        svg.setFill(colorPicker.getValue());
        stack.getChildren().addAll(svg, logo);

        colorPicker.setOnAction(new EventHandler() {
            public void handle(Event t) {
                svg.setFill(colorPicker.getValue());
            }
        });

        stage.setScene(scene);
        stage.show();
    }
}
```

In this example, the color selected in the color picker and obtained through the `getValue` method is applied to the `SVGPath` object. This sample produces the output shown in Figure 24–8

Figure 24–8 ColorPickerSample Application



When you work with a color picker, you can obtain the created custom colors by calling the `getCustomColors()` method. It returns an `ObservableList` of the `Color` objects corresponding to the created colors. You cannot upload them to a color picker on the application start. However, you can set one of the custom colors as a `ColorPicker` value (shown in Example 24–4).

Example 24–4 Obtaining the Custom Colors

```
ObservableList<Color> customColors = colorPicker.getCustomColors();
colorPicker.setValue(customColors.get(index));
```

Changing the Appearance of a Color Picker

The default appearance of the color picker control is defined by the `com.sun.javafx.scene.control.skin.ColorPickerSkin` class. To apply an alternative skin to the color pickers in your JavaFX application, redefine the `-fx-skin` property of the `color-picker` CSS class as shown in Example 24–5.

Example 24–5 Setting an Alternative Skin for a Color Picker

```
.color-picker {
    -fx-skin: "CustomSkin";
}
```

Use the `split-button` and `arrow-button` CSS classes to change the appearance of a color picker control in the JavaFX code, as shown in Example 24–6.

Example 24–6 Setting Appearance for a Color Picker

```
//Sets the split-menu-button
colorPicker.getStyleClass().add("split-button");
//Sets the button
colorPicker.getStyleClass().add("button");
```

You can also modify the default style of a color picker and customize its elements with various CSS classes defined in the `caspian` style sheet. To view this file, go to the `\rt\lib` directory under the directory in which the JavaFX SDK is installed. Use the following command to extract the style sheet from the JAR file: `jar -xf jfxrt.jar com/sun/javafx/scene/control/skin/caspian/caspian.css`. See Skinning JavaFX Applications with CSS for more information about CSS classes and properties. Example 24–7 shows how to alter the default background and the label of a color picker.

Example 24–7 Modifying the Default Appearance of a Color Picker

```
.color-picker {
    -fx-background-color: #669999;
    -fx-background-radius: 0 15 15 0;
}
.color-picker .color-picker-label .text {
    -fx-fill: #ccffcc;
}
```

Add these styles to the `ControlStyle.css` file and enable the style sheets in the JavaFX application by using the following code line: `scene.getStylesheets().add("colorpickersample/ControlStyle.css");`, then compile and the run the ColorPickerSample. The color picker should change its appearance as shown in Figure 24–9.

Figure 24–9 Modified Appearance of a Color Picker

Note that the `ColorPicker` class is an extension of the `ComboBoxBase` class and inherits its CSS properties. You can define new styles for the `combo-box-base` CSS style to unify the combo box and the color picker in the ColorPickerSample application. Replace the styles in the ControlStyle.css file with the styles shown in Example 24–8.

Example 24–8 Setting the Combo-Box-Base Styles

```
.tool-bar:horizontal {
    -fx-background-color: #b3e6b3;
}

.combo-box-base  {
    -fx-background-color: null;
}

.combo-box-base:hover  {
    -fx-effect: dropshadow( three-pass-box , rgba(0,0,0,0.6) , 8, 0.0 , 0 , 0 );
}
```

When you compile and run the ColorPickerSample application with the applied styles, the combo box and the color picker look as shown in Figure 24–10.

Figure 24–10 Unified Style of the Combo Box and Color Picker

Related API Documentation

- `ColorPicker`
- `ComboBoxBase`

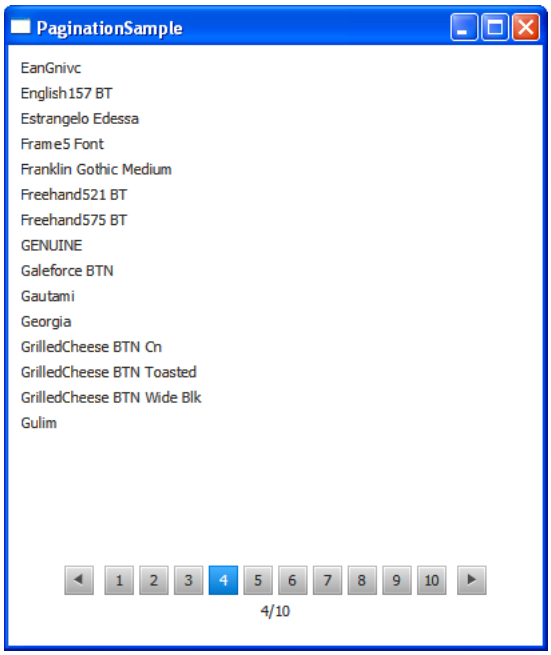
25

Pagination Control

This chapter explains how to add a pagination control to your JavaFX application. It teaches how to add a pagination control to your application, manage its page items, and style the elements of the control with CSS styles.

The pagination control that is used to navigate through multiple pages of content that are divided into smaller parts. Typical uses include navigating through email messages in a mailbox or choosing among search results. In touch devices, a pagination control can be used to browse through single pages of an article or to navigate between screens. Figure 25–1 shows a pagination control that displays the fonts available in an operating system.

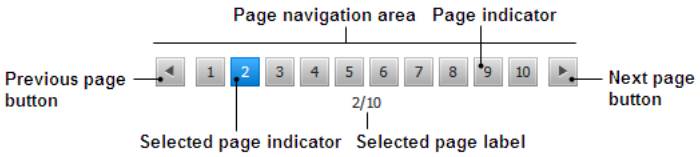
Figure 25–1 Pagination Control



Creating a Pagination Control

A pagination control consists of the page content and the page navigation areas. The Page content area renders and lays out the content according to the application logic. The Page navigation area contains a prefabricated control to preview a particular part of the content. Figure 25–2 shows the structure and basic elements of the navigation area.

Figure 25–2 Navigation Area of a Pagination Control



You can navigate through the pages by clicking a particular page indicator or by clicking the Next page and Previous page buttons. When the first page is selected, the Previous page button is disabled. Similarly, when the last navigation indicator is selected, the Next page button is disabled.

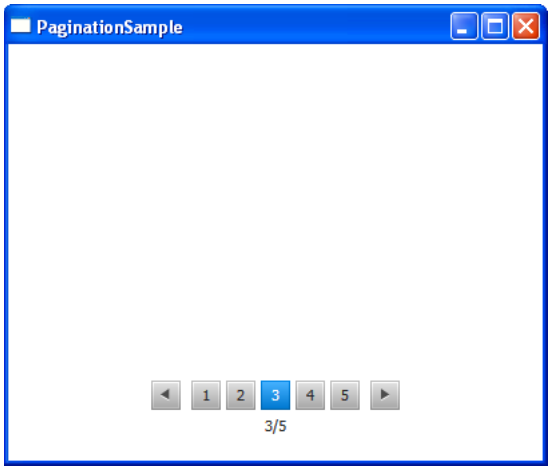
The JavaFX SDK API provides the `Pagination` class to add the pagination control to your JavaFX application. Example 25–1 shows three available constructors of the `Pagination` class.

Example 25–1 Three Constructors of the Pagination Class

```
//Creates a Pagination control with an INDETERMINATE page count
//and the current page index equal to zero
pagination1 = new Pagination();
//Creates a Pagination control with 5 pages
//and the current page index equal to zero
pagination2 = new Pagination(5);
//Creates a Pagination control with 5 pages
//and the current selected index equal to 2
pagination3 = new Pagination(5, 2);
```

The pagination3 control from Example 25–1 is displayed in Figure 25–3.

Figure 25–3 Pagination Control without the Content



Note that page indexes start with 0. Therefore, to start with the third page selected, you need to set the `currentPageIndexProperty` to 2.

The pages of the pagination3 control are empty, because no content is added to the control.

You cannot add any items directly to the pagination control, it requires setting a page factory. Use the `setPageFactory` method of the `Pagination` class to define the page content by implementing a page factory.

Implementing Page Factories

The `setPageFactory` is used to define a page factory for the pagination control. The application developer creates a callback method and sets the pagination page factory to use this callback. The callback method is called when a page has been selected. It loads and returns the content of the selected page. The `null` value must be returned if the selected page index does not exist. Example 25–2 creates a pagination control with 28 pages and populates the pages with the search results, allocating eight items per page.

Example 25–2 Adding Hyperlinks to a Pagination Control

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Pagination;
import javafx.scene.Node;
import javafx.scene.control.Hyperlink;
import javafx.scene.control.Label;
import javafx.scene.layout.AnchorPane;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import javafx.util.Callback;


public class PaginationSample extends Application {
    private Pagination pagination;

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

    public int itemsPerPage() {
        return 8;
    }

    public VBox createPage(int pageIndex) {
        VBox box = new VBox(5);
        int page = pageIndex * itemsPerPage();
        for (int i = page; i < page + itemsPerPage(); i++) {
            VBox element = new VBox();
            Hyperlink link = new Hyperlink("Item " + (i+1));
            link.setVisited(true);
            Label text = new Label("Search results\nfor "+ link.getText());
            element.getChildren().addAll(link, text);
            box.getChildren().add(element);
        }
        return box;
    }

    @Override
    public void start(final Stage stage) throws Exception {
        pagination = new Pagination(28, 0);
        pagination.setStyle("-fx-border-color:red;");
        pagination.setPageFactory(new Callback<Integer, Node>() {
            @Override
```

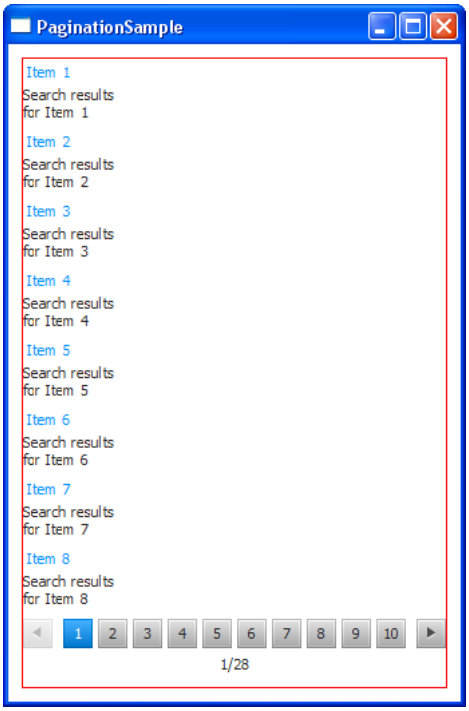
```
                public Node call(Integer pageIndex) {
                    return createPage(pageIndex);
                }
            });

            AnchorPane anchor = new AnchorPane();
            AnchorPane.setTopAnchor(pagination, 10.0);
            AnchorPane.setRightAnchor(pagination, 10.0);
            AnchorPane.setBottomAnchor(pagination, 10.0);
            AnchorPane.setLeftAnchor(pagination, 10.0);
            anchor.getChildren().addAll(pagination);
            Scene scene = new Scene(anchor);
            stage.setScene(scene);
            stage.setTitle("PaginationSample");
            stage.show();
        }
}
```

The number of pages and the selected page are defined within the constructor of the `Pagination` class. Alternatively, you can create a `Pagination` control and set the number of pages and the index of the selected page afterward by using the `setPageCount` and `setCurrentPageIndex` methods.

The content of the `Pagination` control is declared within the `createPage` method that serves as a page factory and is called by the `setPageFactory` method. The `createPage` method creates the pairs of hyperlinks and the corresponding labels, and arranges them vertically, setting a five-pixel interval between the elements.

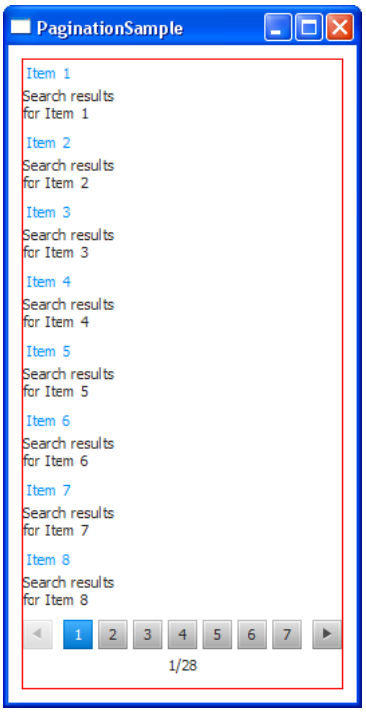
When you compile and run Example 25–2, you should see the output shown in Figure 25–4.

Figure 25–4 Using a Pagination Control to Preview Search Results



The current implementation of the `Pagination` control displays 10 page indicators if the number of pages exceeds 10. To set an alternative value for the displayed page indicators, use the `setMaxPageIndicatorCount` method of the `Pagination` class. For example, add the following line to Example 25–2 to show seven page indicators: `pagination.setMaxPageIndicatorCount(7);`. Figure 25–5 shows the PaginationSample application after the change has been applied.

Figure 25–5 Changing the Number of Page Indicators



Example 25–3 shows another use for the pagination control. The application renders the text fragments, one per page. The number of the fragments is five, and the number of the declared pages of the pagination control is 28. To avoid an `ArrayIndexOutOfBoundsException` condition, add the page index check (highlighted in bold in Example 25–3) and make the callback method return `null` when the number of pages exceeds five.

Example 25–3 Adding Text Snippets to a Pagination Control

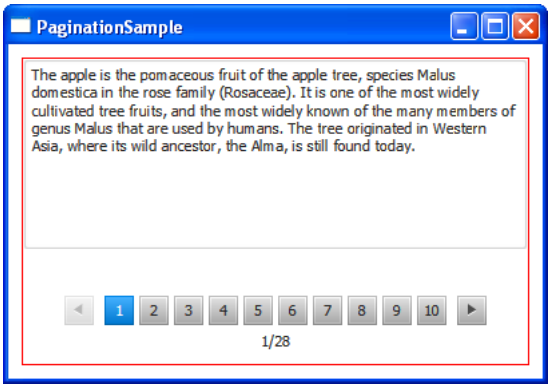
```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Pagination;
import javafx.scene.Node;
import javafx.scene.control.TextArea;
import javafx.scene.layout.AnchorPane;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import javafx.util.Callback;

public class PaginationSample extends Application {

    private Pagination pagination;
    final String[] textPages = new String[]{
        "The apple is the pomaceous fruit of the apple tree, species Malus "
        + "domestica in the rose family (Rosaceae). It is one of the most "
```

```
        + "widely cultivated tree fruits, and the most widely known of "
        + "the many members of genus Malus that are used by humans. "
        + "The tree originated in Western Asia, where its wild ancestor, "
        + "the Alma, is still found today.",
        "The hawthorn is a large genus of shrubs and trees in the rose family,"
        + "Rosaceae, native to temperate regions of the Northern Hemisphere "
        + "in Europe, Asia and North America. The name hawthorn was "
        + "originally applied to the species native to northern Europe, "
        + "especially the Common Hawthorn C. monogyna, and the unmodified "
        + "name is often so used in Britain and Ireland.",
        "The ivy is a flowering plant in the grape family (Vitaceae) native to "
        + " eastern Asia in Japan, Korea, and northern and eastern China. "
        + "It is a deciduous woody vine growing to 30 m tall or more given "
        + "suitable support,  attaching itself by means of numerous small "
        + "branched tendrils tipped with sticky disks.",
        "The quince is the sole member of the genus Cydonia and is native to "
        + "warm-temperate southwest Asia in the Caucasus region. The "
        + "immature fruit is green with dense grey-white pubescence, most "
        + "of which rubs off before maturity in late autumn when the fruit "
        + "changes color to yellow with hard, strongly perfumed flesh.",
        "Aster (syn. Diplopappus Cass.) is a genus of flowering plants "
        + "in the family Asteraceae. The genus once contained nearly 600 "
        + "species in Eurasia and North America, but after morphologic "
        + "and molecular research on the genus during the 1990s, it was "
        + "decided that the North American species are better treated in a "
        + "series of other related genera. After this split there are "
        + "roughly 180 species within the genus, all but one being confined "
        + "to Eurasia."
    };

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

    public int itemsPerPage() {
        return 1;
    }

    public VBox createPage(int pageIndex) {
        VBox box = new VBox(5);
        int page = pageIndex * itemsPerPage();
        for (int i = page; i < page + itemsPerPage(); i++) {
            TextArea text = new TextArea(textPages[i]);
            text.setWrapText(true);
            box.getChildren().add(text);
        }
        return box;
    }

    @Override
    public void start(final Stage stage) throws Exception {
        pagination = new Pagination(28, 0);
        pagination.setStyle("-fx-border-color:red;");
        pagination.setPageFactory(new Callback<Integer, Node>() {

            @Override
            public Node call(Integer pageIndex) {
                if (pageIndex >= textPages.length) {
                    return null;
                } else {
```

```
                    return createPage(pageIndex);
                }
            }
        });

        AnchorPane anchor = new AnchorPane();
        AnchorPane.setTopAnchor(pagination, 10.0);
        AnchorPane.setRightAnchor(pagination, 10.0);
        AnchorPane.setBottomAnchor(pagination, 10.0);
        AnchorPane.setLeftAnchor(pagination, 10.0);
        anchor.getChildren().addAll(pagination);
        Scene scene = new Scene(anchor, 400, 250);
        stage.setScene(scene);
        stage.setTitle("PaginationSample");
        stage.show();
    }
}
```

When you compile and run Example 25–3, you will see the output shown in Figure 25–6.

Figure 25–6 Rendering Text Fragments in a Pagination Control



In some cases you cannot set the exact number of items to render and, therefore, the number of pages in a pagination control. In such situations, you can include a line of code that calculates the number of pages within the constructor of the `Pagination` object. Example 25–4 outputs a list of system fonts and calculates the number of pages as the length of the fonts array divided by the number of items per page.

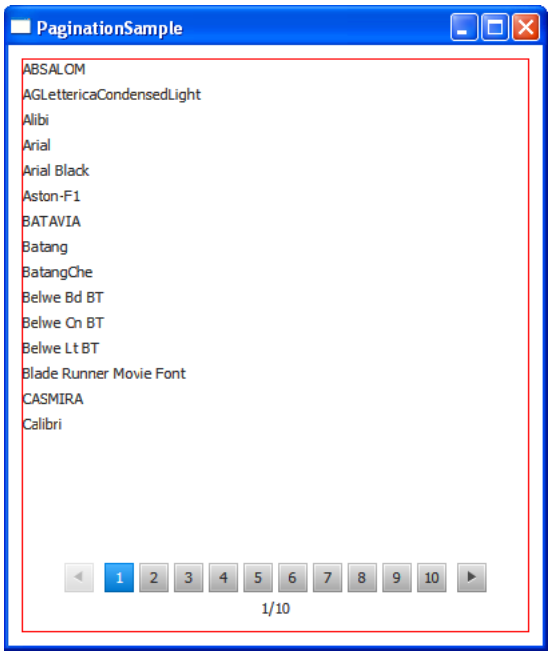
Example 25–4 Adding Content of an Undetermined Size

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Pagination;
import javafx.scene.Node;
import javafx.scene.control.Label;
import javafx.scene.layout.AnchorPane;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.stage.Stage;
import javafx.util.Callback;

public class PaginationSample extends Application {

    private Pagination pagination;
```

```
    String[] fonts = new String[]{};

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

    public int itemsPerPage() {
        return 15;
    }

    public VBox createPage(int pageIndex) {
        VBox box = new VBox(5);
        int page = pageIndex * itemsPerPage();
        for (int i = page; i < page + itemsPerPage(); i++) {
            Label font = new Label(fonts[i]);
            box.getChildren().add(font);
        }
        return box;
    }

    @Override
    public void start(final Stage stage) throws Exception {
        fonts = Font.getFamilies().toArray(fonts);

        pagination = new Pagination(fonts.length/itemsPerPage(), 0);
        pagination.setStyle("-fx-border-color:red;");
        pagination.setPageFactory(new Callback<Integer, Node>() {

            @Override
            public Node call(Integer pageIndex) {
                return createPage(pageIndex);
            }
        });

        AnchorPane anchor = new AnchorPane();
        AnchorPane.setTopAnchor(pagination, 10.0);
        AnchorPane.setRightAnchor(pagination, 10.0);
        AnchorPane.setBottomAnchor(pagination, 10.0);
        AnchorPane.setLeftAnchor(pagination, 10.0);
        anchor.getChildren().addAll(pagination);
        Scene scene = new Scene(anchor, 400, 450);
        stage.setScene(scene);
        stage.setTitle("PaginationSample");
        stage.show();
    }
}
```

When you compile and run this example, it produces the application window shown in Figure 25–7.

Figure 25–7 Using a Pagination Control to Render the System Fonts



Styling a Pagination Control

You can customize the pagination control to display bullet page indicators instead of numeric page indicators by setting the style class `STYLE_CLASS_BULLET`. In addition, you can modify the default pagination styles in the `caspian` style sheet.

Example 25–5 shows some alternative styles for the visual elements of the pagination control in Example 25–4.

Example 25–5 Modified Styles of the Pagination Control

```
.pagination {
    -fx-border-color:  #0E5D79;
}

.pagination .page {
    -fx-background-color: #DDF1F8;
}

.pagination .pagination-control {
    -fx-background-color: #C8C6C6;
}

.pagination .pagination-control .bullet-button {
    -fx-background-color: transparent, #DDF1F8, #0E5D79, white, white;
}

.pagination .pagination-control .bullet-button:selected {
    -fx-background-color: transparent, #DDF1F8, #0E5D79, white, #0E5D79;
}

.pagination .pagination-control .left-arrow, .right-arrow{
    -fx-background-color: #DDF1F8, #0E5D79;
}
```

Example 25–6 applies these styles to the pagination control and sets the bullet style for the page indicators.

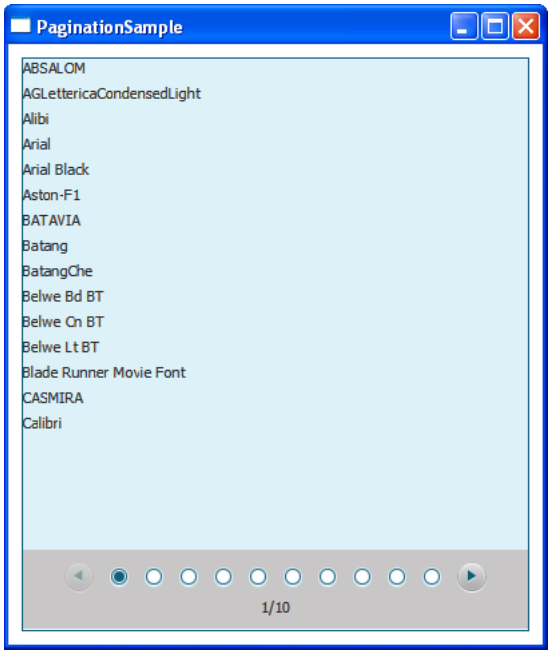
Example 25–6 Enabling the Modified Pagination Control Style in the PaginationSample Application

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Pagination;
import javafx.scene.Node;
import javafx.scene.control.Label;
import javafx.scene.layout.AnchorPane;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.stage.Stage;
import javafx.util.Callback;

public class PaginationSample extends Application {

    private Pagination pagination;
    String[] fonts = new String[]{};

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

    public int itemsPerPage() {
        return 15;
    }

    public VBox createPage(int pageIndex) {
        VBox box = new VBox(5);
        int page = pageIndex * itemsPerPage();
        for (int i = page; i < page + itemsPerPage(); i++) {
            Label font = new Label(fonts[i]);
            box.getChildren().add(font);
        }
        return box;
    }

    @Override
    public void start(final Stage stage) throws Exception {
        fonts = Font.getFamilies().toArray(fonts);

        pagination = new Pagination(fonts.length/itemsPerPage(), 0);
        pagination.getStyleClass().add(Pagination.STYLE_CLASS_BULLET);
        pagination.setPageFactory(new Callback<Integer, Node>() {

            @Override
            public Node call(Integer pageIndex) {
                return createPage(pageIndex);
            }
        });

        AnchorPane anchor = new AnchorPane();
        AnchorPane.setTopAnchor(pagination, 10.0);
        AnchorPane.setRightAnchor(pagination, 10.0);
        AnchorPane.setBottomAnchor(pagination, 10.0);
        AnchorPane.setLeftAnchor(pagination, 10.0);
```

```
        anchor.getChildren().addAll(pagination);
        Scene scene = new Scene(anchor, 400, 450);
        stage.setScene(scene);
        stage.setTitle("PaginationSample");
        scene.getStylesheets().add("paginationsample/ControlStyle.css");
        stage.show();
    }
}
```

When you apply the newly defined styles to the PaginationSample application, and compile and run it, you see the application window shown in Figure 25–8.

Figure 25–8 PaginationSample with Bullet Page Indicators and the New CSS Styles Applied



In addition to the applied styles you can consider the following styles to alter the appearance of the pagination control in your applications:

- `-fx-max-page-indicator-count` — Sets the maximum number of page indicators.
- `-fx-arrows-visible` — Toggles visibility of the Next and Previous button arrows, `true` by default.
- `-fx-tooltip-visible` — Toggles visibility of the page indicator tooltip, `true` by default.
- `-fx-page-information-visible` — Toggles visibility of the page information, `true` by default.
- `-fx-page-information-alignment` — Sets the alignment of the page information.

Related API Documentation

- `Pagination`
- `VBox`

- `AnchorPane`

26

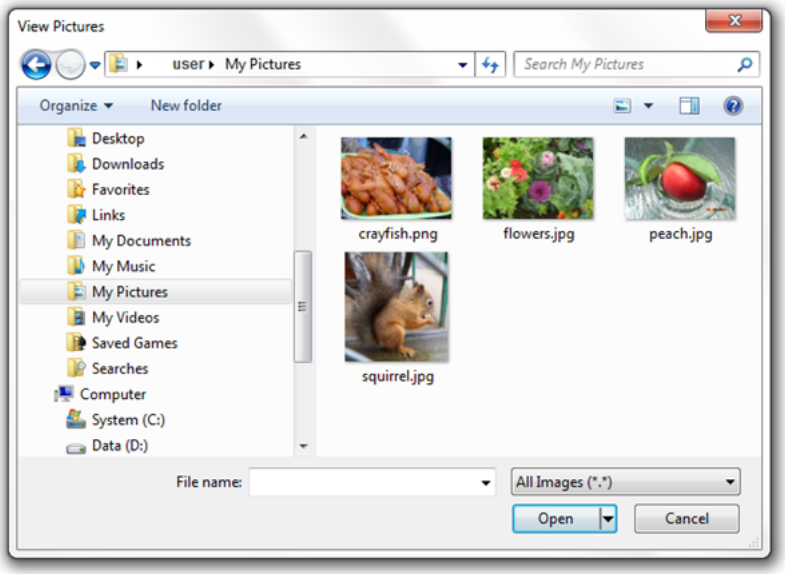
File Chooser

This chapter explains how to use the `FileChooser` class to enable users to navigate the file system. The samples provided in this chapter explain how to open one or several files, configure a file chooser dialog window, and save the application content.

Unlike other user interface component classes, the `FileChooser` class does not belong to the `javafx.scene.controls` package. However, this class deserves to be mentioned in the JavaFX UI Controls tutorial, because it supports one of the typical GUI application functions: file system navigation.

The `FileChooser` class is located in the `javafx.stage` package along with the other basic root graphical elements, such as `Stage`, `Window`, and `Popup`. The View Pictures window in Figure 26–1 is an example of the file chooser dialog in Windows.

Figure 26–1 Example of a File Chooser Window



Opening Files

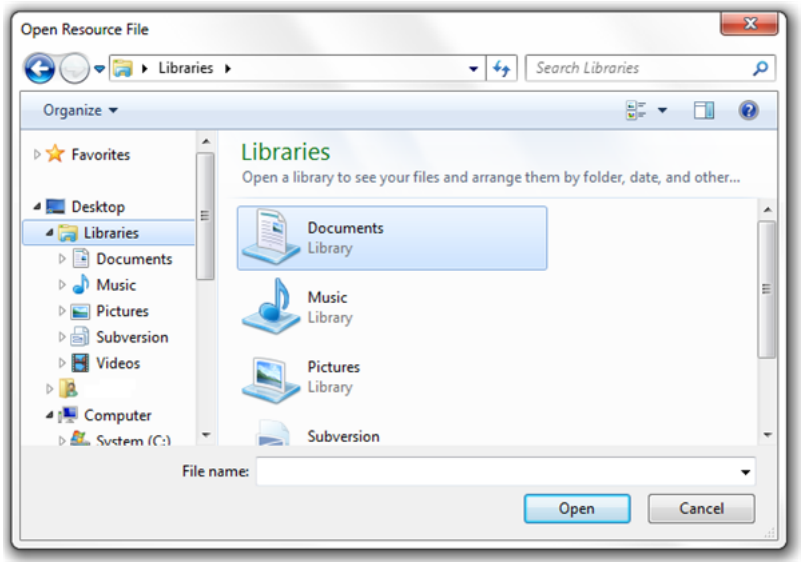
A file chooser can be used to invoke an open dialog window for selecting either a single file or multiple files, and to enable a file save dialog window. To display a file chooser, you typically use the `FileChooser` class. Example 26–1 provides the simplest way to enable a file chooser in your application.

Example 26–1 Showing a File Chooser

```
FileChooser fileChooser = new FileChooser();
fileChooser.setTitle("Open Resource File");
fileChooser.showOpenDialog(stage);
```

After the code from Example 26–1 is added to a JavaFX application, the file chooser dialog window appears immediately when the application starts, as shown in Figure 26–2.

Figure 26–2 Simple File Chooser



Note: Figure 26–2 shows the file chooser in Windows. When you open file choosers in other operating systems that support this functionality, you receive alternative windows. Figure 26–3 and Figure 26–4 show examples of file chooser windows in Linux and Mac OS.

Figure 26–3 File Chooser Window in Linux

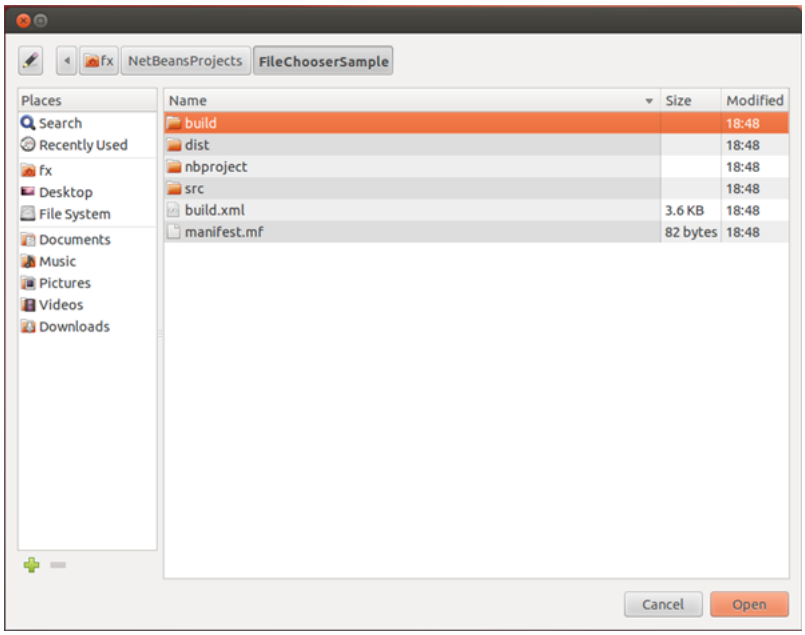
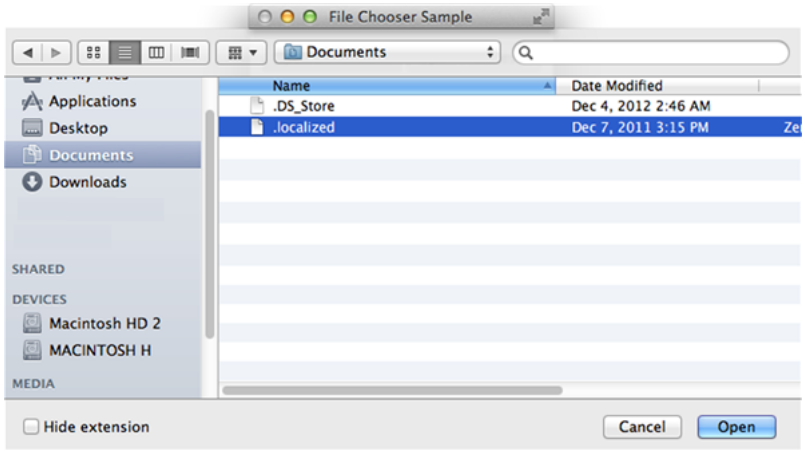


Figure 26–4 File Chooser Window in Mac OS



Although in the previous example the file chooser appears automatically when the application starts, a more typical approach is to invoke a file chooser by selecting the corresponding menu item or clicking a dedicated button. In this tutorial, you create an application that enables a user to click a button and open a one or more pictures located in the file system. Example 26–2 shows the code of the FileChooserSample application that implements this task.

Example 26–2 Opening File Chooser for Single and Multiple Selection

```
import java.awt.Desktop;
import java.io.File;
import java.io.IOException;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;
```

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Pane;
import javafx.scene.layout.VBox;
import javafx.stage.FileChooser;
import javafx.stage.Stage;

public final class FileChooserSample extends Application {

    private Desktop desktop = Desktop.getDesktop();

    @Override
    public void start(final Stage stage) {
        stage.setTitle("File Chooser Sample");

        final FileChooser fileChooser = new FileChooser();

        final Button openButton = new Button("Open a Picture...");
        final Button openMultipleButton = new Button("Open Pictures...");

        openButton.setOnAction(
            new EventHandler<ActionEvent>() {
                @Override
                public void handle(final ActionEvent e) {
                    File file = fileChooser.showOpenDialog(stage);
                    if (file != null) {
                        openFile(file);
                    }
                }
            });

        openMultipleButton.setOnAction(
            new EventHandler<ActionEvent>() {
                @Override
                public void handle(final ActionEvent e) {
                    List<File> list =
                        fileChooser.showOpenMultipleDialog(stage);
                    if (list != null) {
                        for (File file : list) {
                            openFile(file);
                        }
                    }
                }
            });


        final GridPane inputGridPane = new GridPane();

        GridPane.setConstraints(openButton, 0, 0);
        GridPane.setConstraints(openMultipleButton, 1, 0);
        inputGridPane.setHgap(6);
        inputGridPane.setVgap(6);
        inputGridPane.getChildren().addAll(openButton, openMultipleButton);

        final Pane rootGroup = new VBox(12);
```

```
        rootGroup.getChildren().addAll(inputGridPane);
        rootGroup.setPadding(new Insets(12, 12, 12, 12));

        stage.setScene(new Scene(rootGroup));
        stage.show();
    }

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

    private void openFile(File file) {
        try {
            desktop.open(file);
        } catch (IOException ex) {
            Logger.getLogger(
                FileChooserSample.class.getName()).log(
                    Level.SEVERE, null, ex
                );
        }
    }
}
```

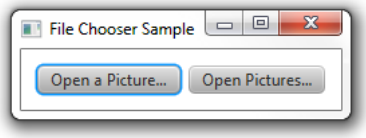
In Example 26–2, the Open a Picture button enables the user to open a file chooser for a single selection, and the Open Pictures button enables the user to open a file chooser for multiple selections. The `setOnAction` methods for these buttons are almost identical. The only difference is in the method that is used to invoke a `FileChooser`.

- The `showOpenDialog` method shows a new file open dialog in which one file can be selected. The method returns the value that specifies the file chosen by the user or `null` if no selection has been made.
- The `showOpenMultipleDialog` method shows a new file open dialog in which multiple files can be selected. The method returns the value that specifies the list of files chosen by the user or `null` if no selection has been made. The returned list cannot be modified and throws `UnsupportedOperationException` on each modification attempt.

Both methods do not return results until the displayed open dialog window is dismissed (in other words, until a user commits or cancels the selection).

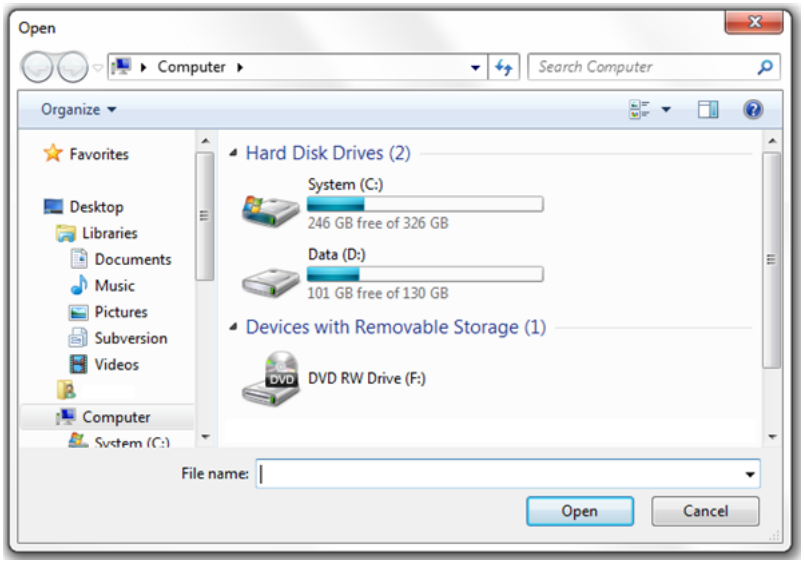
When you compile and run the FileChooserSample application, it produces the window shown in Figure 26–5.

Figure 26–5 FileChooserSample with Two Buttons



When you click either of the buttons, the dialog window shown in Figure 26–6 appears. The opened file chooser dialog window shows the location that is the default for your operating system.

Figure 26–6 Default File Chooser Window



Users of the FileChooserSample application may navigate to a directory containing pictures and select a picture. After a file is selected, it is opened with the associated application. The example code implements this by using the `open` method of the `java.awt.Desktop` class: `desktop.open(file);`.

Note: Availability of the `Desktop` class is platform dependent. Refer to API documentation for more information on the `Desktop` class. You can also use the `isDesktopSupported()` method to check if it is supported in your system.

You can improve the user experience for this application by setting the file chooser directory to the specific directory that contains the pictures.

Configuring a File Chooser

You can configure the file chooser dialog window by setting the `initialDirectory` and `title` properties of a `FileChooser` object. Example 26–3 shows how to specify the initial directory and a suitable title to preview and open pictures.

Example 26–3 Setting the Initial Directory and Window Title

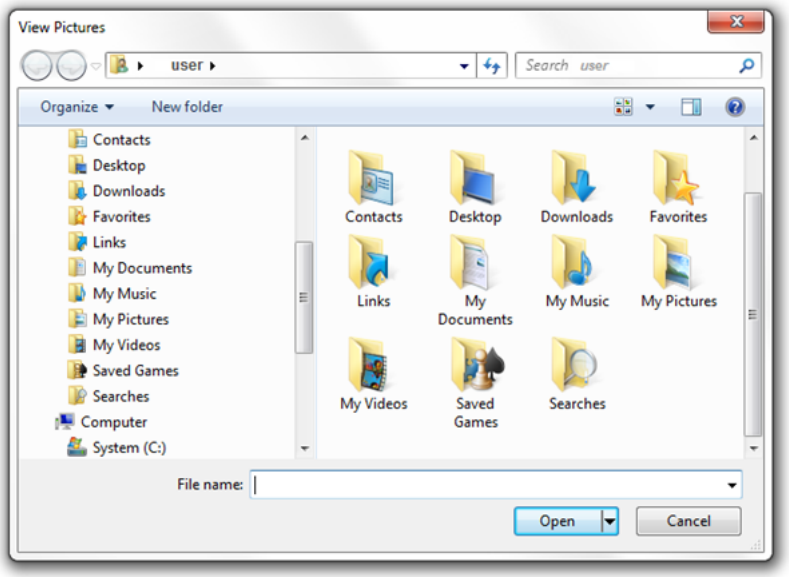
```
import java.awt.Desktop;
import java.io.File;
import java.io.IOException;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Button;
```

```
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Pane;
import javafx.scene.layout.VBox;
import javafx.stage.FileChooser;
import javafx.stage.Stage;

public final class FileChooserSample extends Application {

    private Desktop desktop = Desktop.getDesktop();

    @Override
    public void start(final Stage stage) {
        stage.setTitle("File Chooser Sample");

        final FileChooser fileChooser = new FileChooser();

        final Button openButton = new Button("Open a Picture...");
        final Button openMultipleButton = new Button("Open Pictures...");

        openButton.setOnAction(
            new EventHandler<ActionEvent>() {
                @Override
                public void handle(final ActionEvent e) {
                    configureFileChooser(fileChooser);
                    File file = fileChooser.showOpenDialog(stage);
                    if (file != null) {
                        openFile(file);
                    }
                }
            });

        openMultipleButton.setOnAction(
            new EventHandler<ActionEvent>() {
                @Override
                public void handle(final ActionEvent e) {
                    configureFileChooser(fileChooser);
                    List<File> list =
                        fileChooser.showOpenMultipleDialog(stage);
                    if (list != null) {
                        for (File file : list) {
                            openFile(file);
                        }
                    }
                }
            });

        final GridPane inputGridPane = new GridPane();

        GridPane.setConstraints(openButton, 0, 0);
        GridPane.setConstraints(openMultipleButton, 1, 0);
        inputGridPane.setHgap(6);
        inputGridPane.setVgap(6);
        inputGridPane.getChildren().addAll(openButton, openMultipleButton);

        final Pane rootGroup = new VBox(12);
        rootGroup.getChildren().addAll(inputGridPane);
        rootGroup.setPadding(new Insets(12, 12, 12, 12));

        stage.setScene(new Scene(rootGroup));
        stage.show();
```

```
    }

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

    private static void configureFileChooser(final FileChooser fileChooser){
        fileChooser.setTitle("View Pictures");
        fileChooser.setInitialDirectory(
            new File(System.getProperty("user.home"))
        );
    }


    private void openFile(File file) {
        try {
            desktop.open(file);
        } catch (IOException ex) {
            Logger.getLogger(
                FileChooserSample.class.getName()).log(
                    Level.SEVERE, null, ex
                );
        }
    }
}
```

The `configureFileChooser` method sets the View Pictures title and the path to the user home directory with the My Pictures sub-directory. When you compile and run the FileChooserSample and click one of the buttons, the file chooser shown in Figure 26–7 appears.

Figure 26–7 Opening a Pictures Library



You can also let the users specify the target directory by using the `DirectoryChooser` class. In the code fragment shown in Example 26–4, the click of the `browseButton` invokes the `directoryChooser.showDialog` method.

Example 26–4 Use of the DirectoryChooser Class

```
final Button browseButton = new Button("...");
browseButton.setOnAction(
    new EventHandler<ActionEvent>() {
        @Override
        public void handle(final ActionEvent e) {
            final DirectoryChooser directoryChooser =
                new DirectoryChooser();
            final File selectedDirectory =
                    directoryChooser.showDialog(stage);
            if (selectedDirectory != null) {
                selectedDirectory.getAbsolutePath();
            }
        }
    }
);
```

After the selection is performed, you can assign the corresponding value to the file chooser as follows: `fileChooser.setInitialDirectory(selectedDirectory); .`

Setting Extension Filters

As the next configuration option, you can set the extension filter to determine which files to open in a file chooser, as shown in Example 26–5.

Example 26–5 Setting an Image Type Filter

```
import java.awt.Desktop;
import java.io.File;
import java.io.IOException;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Pane;
import javafx.scene.layout.VBox;
import javafx.stage.FileChooser;
import javafx.stage.Stage;

public final class FileChooserSample extends Application {

    private Desktop desktop = Desktop.getDesktop();

    @Override
    public void start(final Stage stage) {
        stage.setTitle("File Chooser Sample");

        final FileChooser fileChooser = new FileChooser();
        final Button openButton = new Button("Open a Picture...");
        final Button openMultipleButton = new Button("Open Pictures...");

        openButton.setOnAction(
            new EventHandler<ActionEvent>() {
```

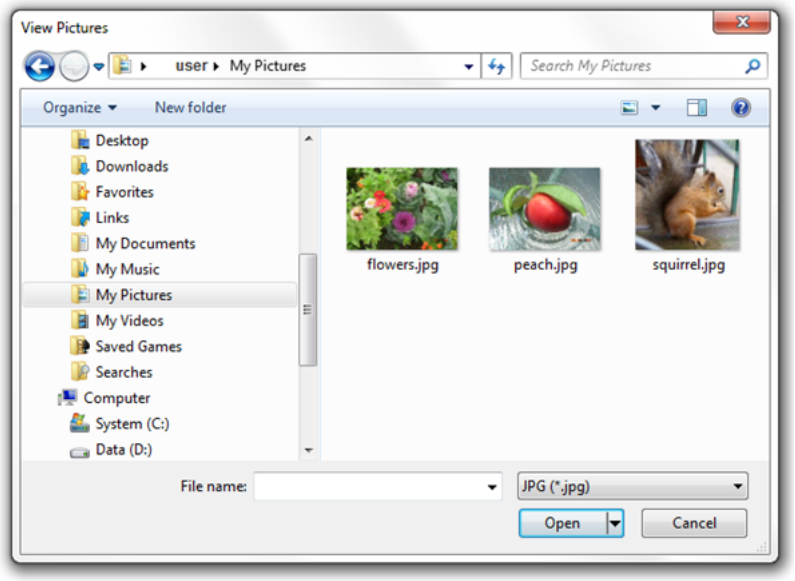
```
            @Override
            public void handle(final ActionEvent e) {
                configureFileChooser(fileChooser);
                File file = fileChooser.showOpenDialog(stage);
                if (file != null) {
                    openFile(file);
                }
            }
        });

    openMultipleButton.setOnAction(
        new EventHandler<ActionEvent>() {
            @Override
            public void handle(final ActionEvent e) {
               configureFileChooser(fileChooser);
                List<File> list =
                    fileChooser.showOpenMultipleDialog(stage);
                if (list != null) {
                    for (File file : list) {
                        openFile(file);
                    }
                }
            }
        });


    final GridPane inputGridPane = new GridPane();

    GridPane.setConstraints(openButton, 0, 1);
    GridPane.setConstraints(openMultipleButton, 1, 1);
    inputGridPane.setHgap(6);
    inputGridPane.setVgap(6);
    inputGridPane.getChildren().addAll(openButton, openMultipleButton);

    final Pane rootGroup = new VBox(12);
    rootGroup.getChildren().addAll(inputGridPane);
    rootGroup.setPadding(new Insets(12, 12, 12, 12));

    stage.setScene(new Scene(rootGroup));
    stage.show();
}

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

private static void configureFileChooser(
    final FileChooser fileChooser) {
        fileChooser.setTitle("View Pictures");
        fileChooser.setInitialDirectory(
            new File(System.getProperty("user.home"))
        );
        fileChooser.getExtensionFilters().addAll(
            new FileChooser.ExtensionFilter("All Images", "*.*"),
            new FileChooser.ExtensionFilter("JPG", "*.jpg"),
            new FileChooser.ExtensionFilter("PNG", "*.png")
        );
}

private void openFile(File file) {
```

```
            try {
                desktop.open(file);
            } catch (IOException ex) {
                Logger.getLogger(FileChooserSample.class.getName()).log(
                    Level.SEVERE, null, ex
                );
            }
        }
    }
```

In Example 26–5, you set an extension filter by using `FileChooser.ExtensionFilter` to define the following options for file selection: All images, JPG, and PNG.

When you compile, run the FileChooserSample code from Example 26–5, and click one of the buttons, the extension filters appear in the file chooser window. If a user selects JPG, then the file chooser displays only pictures of the JPG type. Figure 26–8 captures the moment of selection JPG images in the My Pictures directory.

Figure 26–8 Filtering JPG Files in File Chooser



Saving Files

In addition to opening and filtering files, the `FileChooser` API provides a capability to let a user specify a file name (and its location in the file system) for a file to be saved by the application. The `showSaveDialog` method of the `FileChooser` class opens a save dialog window. Like other show dialog methods, the `showSaveDialog` method returns the file chosen by the user or `null` if no selection has been performed.

The code fragment shown in Example 26–6 is an addition to the Menu sample. It implements one more item of the context menu that saves the displayed image in the file system.

Example 26–6 Saving an Image with the FileChooser Class

```
MenuItem cmItem2 = new MenuItem("Save Image");
    cmItem2.setOnAction(new EventHandler<ActionEvent>() {
        public void handle(ActionEvent e) {
```

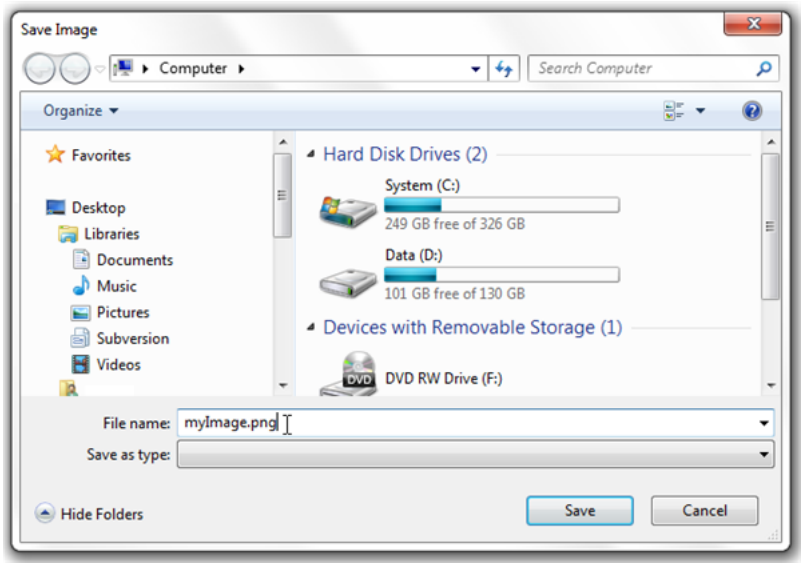
```
                FileChooser fileChooser = new FileChooser();
                fileChooser.setTitle("Save Image");
                System.out.println(pic.getId());
                File file = fileChooser.showSaveDialog(stage);
                if (file != null) {
                    try {
                        ImageIO.write(SwingFXUtils.fromFXImage(pic.getImage(),
                            null), "png", file);
                    } catch (IOException ex) {
                        System.out.println(ex.getMessage());
                    }
                }
            }
        }
);
```

When you add Example 26–6 to the MenuSample application (find the source code in the Application Files), compile and run it, you enable the Save Image menu item, as shown in Figure 26–9.

Figure 26–9 Saving Image

After a user selects the Save Image item, the Save Image window shown in Figure 26–10 appears.

Figure 26–10 The Save Image Window



The Save Image window corresponds to the typical user experience for the save dialog windows: the user needs to select the target directory, type in the name of the saving file, and click Save.

Related API Documentation

- `FileChooser`
- `DirectoryChooser`

27

Customization of UI Controls

This chapter describes the aspects of UI control customization and summarizes some tips and tricks provided by Oracle to help you modify the appearance and behavior of UI controls.

You can learn how to customize the controls from the sample applications in the UIControlSamples project by applying Cascading Style Sheets (CSS), redefining the default behavior, and using cell factories. For more specific cases, when the task of your application requires unique features that cannot be implemented with the classes of the `javafx.scene.control` package, extend the `Control` class to invent your own control.

Applying CSS

You can change the look of UI controls by redefining the style definitions of the JavaFX `caspian` style sheets. Skinning JavaFX Applications with CSS explains the general concepts and approaches to modifying the styles and enabling them in a JavaFX application.

Consider some of the specific tasks that are frequently requested by developers at the JavaFX forum.

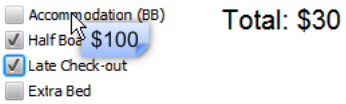
Although the `Tooltip` class does not have any properties or methods to change the default color of the tooltip, you can modify the `-fx-background-color` property of the `.tooltip` CSS class as shown in Example 27–1.

Example 27–1 Changing the Background Color of a Tooltip

```
.tooltip {
    -fx-background-color: linear-gradient(#e2ecfe, #99bcfd);
}
.page-corner {
    -fx-background-color: linear-gradient(from 0% 0% to 50% 50%,#3278fa,#99bcfd);
}
```

The `.page-corner` CSS class defines the color of the right-bottom corner of the tooltip. When you add the code in Example 27–1 to the style sheets of the TooltipSample and apply the style sheets to the scene, the tooltip changes its color to blue. See Figure 27–1 to evaluate the effect.

Figure 27–1 Tooltip with the Blue Background Color



Note that when you modify the default style of a tooltip, the new look is applied to all the tooltips in your application.

Another popular design task is changing the default marks for the controls. For example, the default style of the `CheckBox` class defines the traditional check mark for the selected state. You can redefine the shape of the mark as well as its color as shown in Example 27–2.

Example 27–2 Alternative Mark for a Checkbox

```
.check-box .mark {
    -fx-shape:
"M2,0L5,4L8,0L10,0L10,2L6,5L10,8L10,10L8,10L5,6L2,10L0,10L0,8L4,5L0,2L0,0Z";
}
.check-box:selected .mark {
    -fx-background-color: #0181e2;
}
```

The `-fx-shape` property sets the new SVG path for the mark, and the `-fx-background-color` property defines its color. When the modified style sheets are enabled in the CheckBoxSample application, the selected checkboxes contain X marks instead of check marks, as shown in Figure 27–2.

Figure 27–2 ComboBoxSample with the Modified Checkbox Style

Many developers asked how to overcome the limitation in visual style of the `TableView` and `ListView` controls. By default, all rows in these controls are shown, whether they are empty or not. With the proper CSS settings, you can set a specific color for all empty rows. Example 27–3 implements this task for a `TableView` control.

Example 27–3 Setting Color for Empty Rows in a Table View

```
.table-row-cell:empty {
    -fx-background-color: lightyellow;
}

.table-row-cell:empty .table-cell {
    -fx-border-width: 0px;
}
```

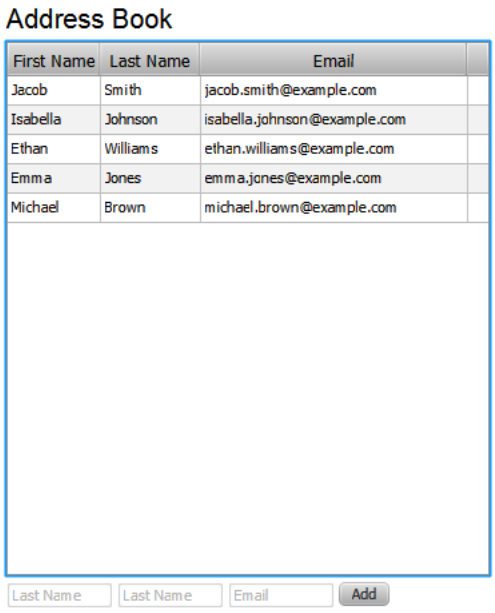
The first CSS style determines that all empty rows, regardless of whether they are even or odd, should have light yellow backgrounds. When the table-row-cell is empty, the second CSS statement removes the vertical border that is painted on the right-hand side of all table cells.

When the CSS styles from Example 27–3 are enabled in the TableViewSample application, the Address Book table looks as shown Figure 27–3.

Figure 27–3 TableViewSample with Color Added to the Empty Rows

You can even set the `null` value for the background color of the empty cells. The style sheets will use the default background color of the table view in this case. See Figure 27–4 to evaluate the effect.

Figure 27–4 TableViewSample with Null Background Color Added to the Empty Rows



You can set more CSS properties for UI Controls to alter their shapes, color schemes, and the applied effects. See the JavaFX CSS Reference Guide for more information about available CSS properties and classes.

Altering Default Behavior

Many developers requested a specific API to restrict input in the text field, for example, to allow only number values. Example 27–4 provides a simple application with a numeric text field.

Example 27–4 Prohibiting Letters in the Text Field

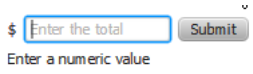
```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;

public class CustomTextFieldSample extends Application {

    final static Label label = new Label();

    @Override
    public void start(Stage stage) {
        Group root = new Group();
        Scene scene = new Scene(root, 300, 150);
        stage.setScene(scene);
        stage.setTitle("Text Field Sample");

        GridPane grid = new GridPane();
        grid.setPadding(new Insets(10, 10, 10, 10));
        grid.setVgap(5);
        grid.setHgap(5);

        scene.setRoot(grid);
        final Label dollar = new Label("$");
        GridPane.setConstraints(dollar, 0, 0);
        grid.getChildren().add(dollar);

        final TextField sum = new TextField() {
            @Override
            public void replaceText(int start, int end, String text) {
                if (!text.matches("[a-z, A-Z]")) {
                    super.replaceText(start, end, text);
                }
                label.setText("Enter a numeric value");
            }

            @Override
            public void replaceSelection(String text) {
                if (!text.matches("[a-z, A-Z]")) {
                    super.replaceSelection(text);
                }
            }
```

```
        };

        sum.setPromptText("Enter the total");
        sum.setPrefColumnCount(10);
        GridPane.setConstraints(sum, 1, 0);
        grid.getChildren().add(sum);

        Button submit = new Button("Submit");
        GridPane.setConstraints(submit, 2, 0);
        grid.getChildren().add(submit);

        submit.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent e) {
                label.setText(null);
            }
        });

        GridPane.setConstraints(label, 0, 1);
        GridPane.setColumnSpan(label, 3);
        grid.getChildren().add(label);

        scene.setRoot(grid);
        stage.show();
    }

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

To redefine the default implementation of the `TextField` class, you must override the `replaceText` and `replaceSelection` methods inherited from the `TextInputControl` class.

When the user tries to enter any letter in the Sum text field, no symbols appear, and the warning message is shown. Figure 27–5 illustrates this situation.

Figure 27–5 Attempt to Enter Letter Symbols



However, when the user attempts to enter the numeric values, they appear in the field as shown in Figure 27–6.

Figure 27–6 Entering Numeric Values

Implementing Cell Factories

Appearance and even behavior of four UI controls can be entirely customized by using the mechanism of cell factories. You can apply cell factories to `TableView`, `ListView`, `TreeView`, and `ComboBox`. A cell factory is used to generate cell instances, which are used to represent any single item of these controls.

The `Cell` class extends the `Labeled` class, which provides all the required properties and methods to implement the most typical use case — showing and editing text. However, when the task of your application requires showing graphical objects in the lists or tables, you can use the `graphic` property and place any `Node` in the cell (see the Cell class API specification for more information about custom cells).

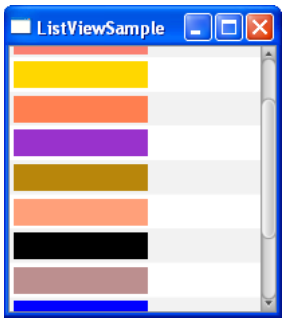
For instance, the code fragments in Example 27–5 create a cell factory for the list view and redefine the content of the cells within the `updateItem` method, so that the list shows rectangles of different colors.

Example 27–5 Implementing Cell Factories for the ListView Control

```
list.setCellFactory(new Callback<ListView<String>, ListCell<String>>() {
    Override public ListCell<String> call(ListView<String> list) {
        return new ColorRectCell();
    }
});
...
static class ColorRectCell extends ListCell<String> {
    @Override
    public void updateItem(String item, boolean empty) {
        super.updateItem(item, empty);
        Rectangle rect = new Rectangle(100, 20);
        if (item != null) {
            rect.setFill(Color.web(item));
            setGraphic(rect);
        } else {
            setGraphic(null);
        }
    }
}
```

Figure 27–7 shows how this customized list looks in the ListViewSample of the UIControlSamples project.

Figure 27–7 List View with Color Rectangles



This tutorial uses the cell factory mechanism extensively to customize UI controls. Table 27–1 summarizes the coding templates that you can use to implement cell factories on your applications.

Table 27–1 Cell Factory Coding Patterns

Control	Coding Pattern
ListView, ComboBox	`list.setCellFactory(new Callback<ListView<String>, ListCell<String>>() { @Override public ListCell<String> call(ListView<String> list) { //cell implementation } });`
TableView	`column.setCellFactory(new Callback<TableColumn, TableCell>() { public TableCell call(TableColumn p) { //cell implementation } });`
TreeView	`tree.setCellFactory(new Callback<TreeView<String>, TreeCell<String>>(){ @Override public TreeCell<String> call(TreeView<String> p) { //cell implementation } });`

You can customize these controls by using the cell factory mechanism or use the prefabricated cell editor implementations that provide specific data models underlying the visualization. Table 27–2 lists the corresponding classes available in the JavafX API.

Table 27–2 Cell Editor Classes for the List View, Tree View, and Table View Controls

Control	Cell Editor Classes
List view	■ `CheckBoxListCell` ■ `ChoiceBoxListCell` ■ `ComboBoxListCell` ■ `TextFieldListCell`
Tree view	■ `CheckBoxTreeCell` ■ `ChoiceBoxTreeCell` ■ `ComboBoxTreeCell` ■ `TextFieldTreeCell`
Table view	■ `CheckBoxTableCell` ■ `ChoiceBoxTableCell` ■ `ComboBoxTableCell` ■ `ProgressBarTableCell` ■ `TextFieldTableCell`

Each cell editor class draws a specific node inside the cell. For example, the `CheckBoxListCell` class draws a `CheckBox` node inside the list cell.

To evaluate more cell factory and cell editor use cases, see the Table View, Tree View, and Combo Box chapters.

Related Documentation and Resources

- Skin Applications with CSS
- JavaFX CSS Reference Guide
- JavaFX News, Demos, and Insight