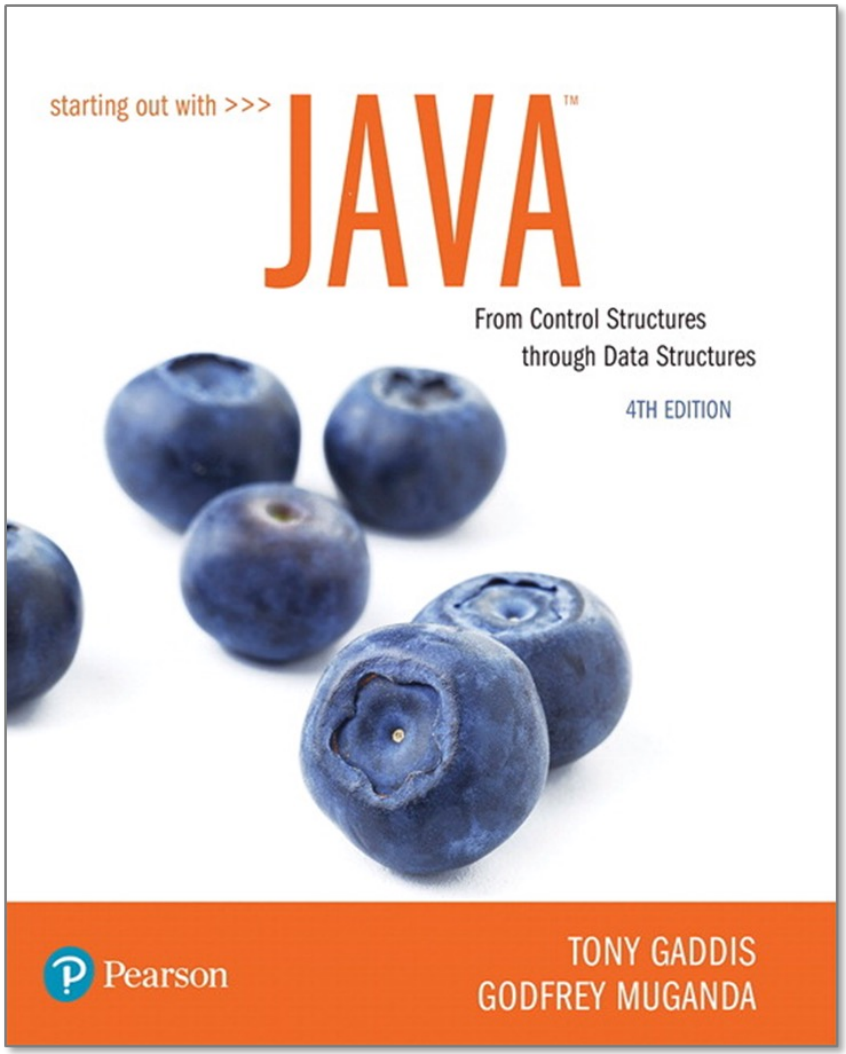
STARTING OUT WITH JAVA™

4th Edition



Chapter 13

JavaFX: Advanced Controls

Topics

- Styling JavaFX Applications with CSS
- `RadioButton` Controls
- `CheckBox` Controls
- `ListView` Controls
- `ComboBox` Controls
- `Slider` Controls
- `TextArea` Controls
- Menus
- The `FileChooser` Class
- Using Console Output to Debug a GUI Application

Styling JavaFX Applications with CSS (1 of 3)

- CSS: Cascading Style Sheets
- A simple language that specifies how a user interface should appear
- CSS was originally created for the Web
- You can use CSS to style a JavaFX user interface

Styling JavaFX Applications with CSS (2 of 3)

- You typically create one or more *style sheets* to accompany your JavaFX application.
- A style sheet is a text file containing one or more style definitions, written in the following general format:

```
selector {
    property: value;
    property: value;
}
```

Styling JavaFX Applications with CSS (3 of 3)

- *selector* – a name that determines the node or nodes that are affected by the style definition
- *property* – the name of a property
- *value* – a value to assign to the property

```
selector {
    property: value;
    property: value;
}
```

Example Style Definition (1 of 2)

- `.label` – specifies that this style definition applies to `Label` controls
- The style rule specifies that the `-fx-font-size` property should be set to 20pt.
- As a result, this style definition specifies that `Label` controls should display their text in a 20-point font.

```
.label {
   -fx-font-size: 20pt;
}
```

Example Style Definition (2 of 2)

- This style definition specifies that `Label` controls should display their text in a cursive, 14-point, italic, bold font, with a dotted border around the control.

```
.label {
   -fx-font-family: cursive;
   -fx-font-size: 14pt;
   -fx-font-style: italic;
   -fx-font-weight: bold;
   -fx-border-style: dotted;
}
```

Type Selectors

Class Name in JavaFX	Corresponding Type Selector
Button	.button
CheckBox	.check-box
CheckMenuItem	.check-menu-item
ComboBox	.combo-box
ImageView	.image-view
Label	.label
ListView	.list-view
Menu	.menu
MenuBar	.menu-bar
MenuItem	.menu-item
RadioButton	.radio-button
RadioMenuItem	.radio-menu-item
Slider	.slider
TextArea	.text-area
TextField	.text-field

Properties

- JavaFX supports a large number of CSS properties.
- Part of the process of creating stylesheets is determining which properties are available for the nodes in your application.
- Oracle publishes a comprehensive CSS reference guide documenting all the available properties: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html

Applying a Stylesheet to a JavaFX Application (1 of 2)

- A stylesheet is a text file containing style definitions.
- Stylesheets are usually saved with the `.css` file extension.
- To apply a stylesheet to a JavaFX application:
  - Save the stylesheet in the same directory as the JavaFX application
  - Use the scene object's `getStylesheets().add()` method to apply the stylesheet

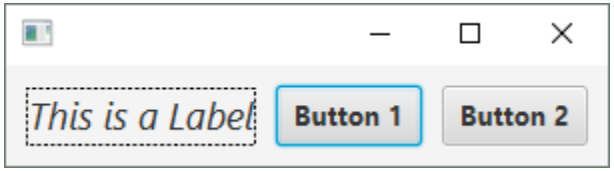
Applying a Stylesheet to a JavaFX Application (2 of 2)

- Example:
  - We have a stylesheet named `mystyles.css`
  - In our Java code, the `scene` variable references the Scene object
  - We would use the following statement to add the stylesheet to the scene:

```
scene.getStylesheets().add("mystyles.css");
```

demo2.css

```
1  .label {
2     -fx-font-family: cursive;
3     -fx-font-size: 14pt;
4     -fx-font-style: italic;
5     -fx-border-style: dashed;
6  }
7
8  .button {
9     -fx-font-size: 10pt;
10    -fx-font-weight: 900;
11 }
```




```
@Override
public void start(Stage primaryStage)
{
   // Create a Label and some Buttons.
   Label label = new Label("This is a Label");
   Button button1 = new Button("Button 1");
   Button button2 = new Button("Button 2");

   // Put the controls in an HBox.
   HBox hbox = new HBox(10, label, button1, button2);
   hbox.setAlignment(Pos.CENTER);
   hbox.setPadding(new Insets(10));

   // Create a Scene and display it.
   Scene scene = new Scene(hbox);
   scene.getStylesheets().add("demo2.css");
   primaryStage.setScene(scene);
   primaryStage.show();
}
```

Applying Styles to the Root Node

- To apply styles to all of the nodes in a scene, use the `.root` selector.

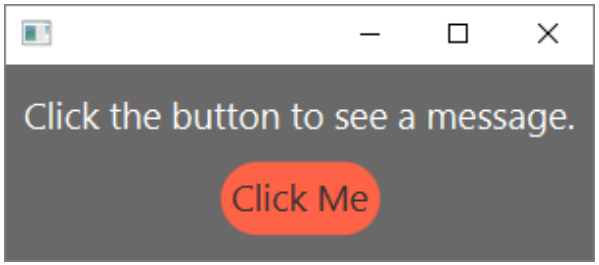
```
.root {
    -fx-background-color: dimgrey;
    -fx-font-size: 14pt;
}
```

mystyle.css

```
.root {
    -fx-background-color: dimgrey;
    -fx-font-size: 14pt;
}

.label {
    -fx-text-fill: white;
}

.button {
    -fx-background-color: tomato;
    -fx-border-radius: 20;
    -fx-background-radius: 20;
    -fx-padding: 5;
}
```




```
// Put the Label and Button in a Vbox...
VBox vbox = new VBox(10, myLabel, myButton);
vbox.setAlignment(Pos.CENTER);

// Create a Scene and display it.
Scene scene = new Scene(vbox, 300, 100);
scene.getStylesheets().add("mystyle.css");
primaryStage.setScene(scene);
primaryStage.show();
```

Working with RGB Colors (1 of 7)

- An RGB color is specified in the following format:

  ```
  #RRGGBB
  ```

  Where RR, GG, and BB are hexadecimal numbers (in the range of 0-255) representing the intensity of the color's red, green, and blue channels.

Working with RGB Colors (2 of 7)

- Example:

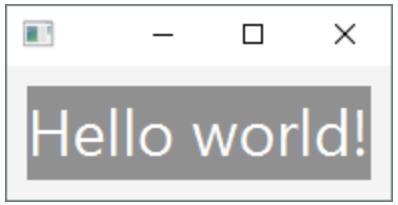
  `#007fff`

  - `00` is the red intensity
  - `7f` is the green intensity
  - `ff` is the blue intensity

Working with RGB Colors (3 of 7)

demo6.css

```
1  .label {
2     -fx-font-size: 24pt;
3     -fx-text-fill: #FFFFFF;
4     -fx-background-color: #909090;
5  }
```



Working with RGB Colors (4 of 7)

- You can also use *functional RGB notation* to specify colors.
- The general format is:

  `rgb(`*`red, green, blue`*`)`

  - The *red*, *green*, and *blue* values can be:
  - an integer in the range of 0 through 255, or
  - a percentage in the range of 0 percent through 100 percent

Working with RGB Colors (5 of 7)

- Example:

```
rgb(200, 0, 100)
```

  - `200` is the red intensity
  - `0` is the green intensity
  - `100` is the blue intensity

Working with RGB Colors (6 of 7)

- Example:

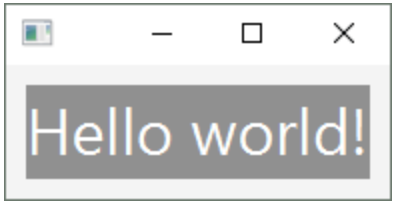
```
rgb(10%, 100%, 50%)
```

  - `10%` is the percentage of red intensity
  - `100%` is the percentage of green intensity
  - `50%` is the percentage of blue intensity

Working with RGB Colors (7 of 7)

demo7.css

```
1  .label {
2     -fx-font-size: 24pt;
3     -fx-text-fill: rgb(255, 255, 255);
4     -fx-background-color: rgb(144, 144, 144);
5  }
```



Named Colors (1 of 3)

- CSS provides numerous predefined names that represent colors.
- These are known as the *named colors*. Here are a few of them:

antiquewhite	aqua	black	blue
chocolate	crimson	darkgray	darkgreen
gold	gray	green	lavender
maroon	olive	orange	plum
purple	red	sandybrown	tan
teal	violet	white	yellow

Named Colors (2 of 3)

- You can use these names just as you would use a hexadecimal color number, or a functional RGB expression.
- JavaFX currently supports 148 named colors.
- See the entire list in Appendix K, or refer to the official CSS reference guide at: https://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html

Named Colors (3 of 3)

- Example:

demo8.css

```
1 .label {
2     -fx-text-fill: white;
3     -fx-background-color: blue;
4 }
```

Creating a Custom Style Class Name (1 of 2)

- Suppose you want some of the Buttons in an application to appear with a black background and white text, and you want other Buttons to appear with a white background and black text.
- You create two custom style classes:
  - one for the black buttons
  - another for the white buttons

Creating a Custom Style Class Name (2 of 2)

- Example: demo9.css

```
1  .button-black {
2      -fx-background-color: black;
3      -fx-text-fill: white;
4  }
5
6  .button-white {
7      -fx-background-color: white;
8      -fx-text-fill: black;
9  }
```

To apply the `.button-black` style definition to a control named `button1`:

```
button1.getStyleClass().add("button-black");
```

ID Selectors (1 of 3)

- Each node in the scene graph can optionally be assigned a unique identifier known as an *ID*.
- A node's ID is a string that can be used to identify the node in a style sheet.
- This is useful when you want to apply a style to one specific node.

ID Selectors (2 of 3)

- You assign an ID to a node with the node's `setId` method:

```
Label outputLabel = new Label("File not found!");
outputLabel.setId("label-error");
```

- This assigns `label-error` as the ID for the `outputLabel` control.

ID Selectors (3 of 3)

- Next, we create a style definition with the selector `#label-error` in our style sheet:

```
#label-error {
   -fx-background-color: red;
   -fx-text-fill: white;

}
```

- The # character indicates this is an ID selector.

- This style definition will be applied only to the node with the ID `label-error`.

Inline Style Rules (1 of 2)

- Style rules can be applied directly to a node in the application's Java code, using the node's `setStyle` method:

```
Label outputLabel = new Label("Hello world!");
outputLabel.setStyle("-fx-font-size: 24pt");
```

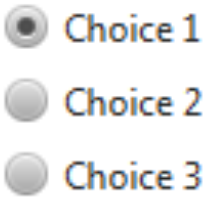
Inline Style Rules (2 of 2)

- To apply multiple style rules to a node, separate them with a semicolon:

```
Label outputLabel = new Label("Hello world!");
outputLabel.setStyle("-fx-font-size: 24pt; -fx-
font-weight: bold");
```

RadioButton Controls (1 of 7)

- RadioButton controls allow the user to select one choice from several possible options.



RadioButton Controls (2 of 7)

- The `RadioButton` class is in the `javafx.scene.control` package.

```
RadioButton radio1 = new RadioButton("Choice 1");
RadioButton radio2 = new RadioButton("Choice 2");
RadioButton radio3 = new RadioButton("Choice 3");
```

RadioButton Controls (3 of 7)

- RadioButton controls are normally grouped together in a *toggle group*.
- Only one of the RadioButton controls in a toggle group may be selected at any time.
- Clicking on a RadioButton selects it and automatically deselects any other RadioButton in the same toggle group.

RadioButton Controls (4 of 7)

- To create a toggle group, you use the `ToggleGroup` class, which is in the `javafx.scene.control` package:

```
ToggleGroup myToggleGroup = new ToggleGroup();
```

- After creating a `ToggleGroup` object, you call each `RadioButton` control's `setToggleGroup` method to add them to the `ToggleGroup`.

RadioButton Controls (5 of 7)

```
// Create some RadioButtons.
RadioButton radio1 = new RadioButton("Option 1");
RadioButton radio2 = new RadioButton("Option 2");
RadioButton radio3 = new RadioButton("Option 3");

// Create a ToggleGroup.
ToggleGroup radioGroup = new ToggleGroup();

// Add the RadioButtons to the ToggleGroup.
radio1.setToggleGroup(radioGroup);
radio2.setToggleGroup(radioGroup);
radio3.setToggleGroup(radioGroup);
```

RadioButton Controls (6 of 7)

- To determine whether a `RadioButton` is selected, you call the `RadioButton` class's `isSelected` method.

```
if (radio1.isSelected())
{
    //  Code here executes if the radio
    //  button is selected.
}
```

RadioButton Controls (7 of 7)

- You usually want one of the `RadioButtons` in a group to be initially selected.

- You can select a `RadioButton` in code with the `RadioButton` class's `setSelected` method:

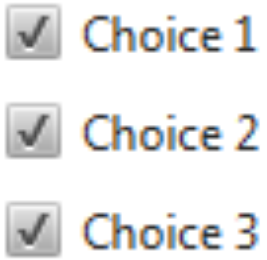
```
radio1.setSelected(true);
```

Responding to `RadioButton` Clicks

- If you want an action to take place immediately when the user clicks a `RadioButton`, register an `ActionEvent` handler with the `RadioButton` control.
- The process is the same as with the `Button` control.
- See `RadioButtonEvent.java` in your textbook.

CheckBox Controls (1 of 4)

- CheckBox controls allow the user to make yes/no or on/off selections.



CheckBox Controls (2 of 4)

- The `CheckBox` class is in the `javafx.scene.control` package.

```
CheckBox choice1 = new CheckBox("Choice 1");
CheckBox choice2 = new CheckBox("Choice 2");
CheckBox choice3 = new CheckBox("Choice 3");
```

CheckBox Controls (3 of 4)

- To determine whether a `CheckBox` is selected, you call the `CheckBox` class's `isSelected` method:

```
if (check1.isSelected())
{
    // Code here executes if the check
    // box is selected.
}
```

CheckBox Controls (4 of 4)

- You can select a `CheckBox` in code with the `CheckBox` class's `setSelected` method:

```
check1.setSelected(true);
```

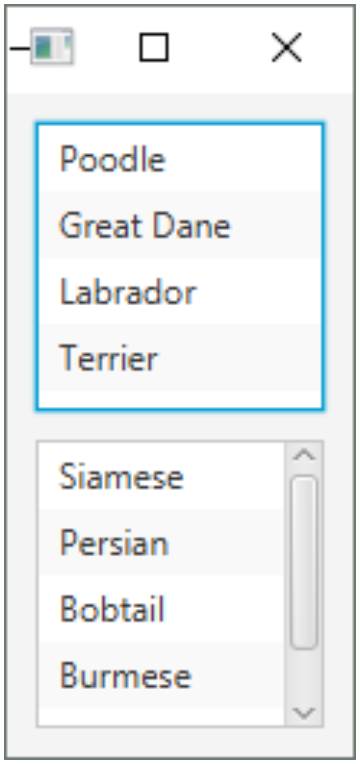
Responding to CheckBox Clicks

- If you want an action to take place immediately when the user clicks a `CheckBox`, register an `ActionEvent` handler with the `CheckBox` control.

- The process is the same as with the `Button` control.

`ListView` Controls (1 of 14)

- The `ListView` control displays a list of items and allows the user to select one or more items from the list.



ListView Controls (2 of 14)

- The `ListView` class is in the `javafx.scene.control` package.

```
ListView<String> dogListView = new ListView<>();
```

- Adding items to the `ListView`:

```
dogListView.getItems().addAll(
   "Poodle", "Great Dane", "Labrador", "Terrier");
```

ListView Controls (3 of 14)

- Creating a `ListView` with a preferred size:

```
ListView<String> catListView = new ListView<>();
catListView.setPrefSize(120, 100);
catListView.getItems().addAll("Siamese", "Persian", "Bobtail",
                              "Burmese", "Tabby");
```

ListView Controls (4 of 14)

- Use the `ListView` class's `getSelectionModel().getSelectedItem()` method get the item that is currently selected.

```
String selected = listView.getSelectionModel().getSelectedItem();
```

- If no item is selected in the `ListView`, the method will return `null`.
- See `ListViewDemo1.java` in your textbook

ListView Controls (5 of 14)

- Each item in a `ListView` control is assigned an index.
- The first item (at the top of the list) has the index 0, the second item has the index 1, and so forth.
- Use the `ListView` class's `getSelectionModel().getSelectedIndex()` method get the index of the item that is currently selected:

```
int index = lististView.getSelectionModel().getSelectedIndex();
```

- If no item is selected in the `ListView`, the method will return `-1`.
- See `ListViewDemo2.java` in your textbook

ListView Controls (6 of 14)

- When the user selects an item in a `ListView`, a *change event* occurs.
- To immediately perform an action when the user selects an item in a ListView, write an event handler that responds to the change event:

```
listView.getSelectionModel().selectedItemProperty().addListener(event ->
{
   // Write event handling code here ...
});
```

- See `ListViewDemo3.java` in your textbook

ListView Controls (7 of 14)

- The `ListView` control's `getItems().addAll()` method adds items to the `ListView`.
- If the `ListView` already contains items, the `getItems().addAll()` method will not erase the existing items, but will add the new items to the existing list.
- If you want to replace the existing items in a `ListView`, use the `getItems().addAll()` method instead.

ListView Controls (8 of 14)

- To initialize a ListView control with the contents of an array or an ArrayList:
  - Convert the array or ArrayList to an ObservableList. (This requires the FXCollections class, in the javafx.collections package)
  - Pass the ObservableList as an argument to the ListView class's constructor.

ListView Controls (9 of 14)

- Example: initializing a `ListView` control with the contents of an array:

```
// Create a String array.
String[] strArray = {"Monday", "Tuesday", "Wednesday"};

// Convert the String array to an ObservableList.
ObservableList<String> strList =
       FXCollections.observableArrayList(strArray);

// Create a ListView control.
ListView<String> listView = new ListView<>(strList);
```

ListView Controls (10 of 14)

- Example: initializing a `ListView` control with the contents of an `ArrayList`:

```
// Create an ArrayList of Strings.
ArrayList<String> strArrayList = new ArrayList<>();
strArrayList.add("Monday");
strArrayList.add("Tuesday");
strArrayList.add("Wednesday");

// Convert the ArrayList to an ObservableList.
ObservableList<String> strList =
       FXCollections.observableArrayList(strArrayList);

// Create a ListView control.
ListView<String> listView = new ListView<>(strList);
```

`ListView` Controls (11 of 14)

- The `ListView` control can operate in either of the following selection modes:
  - Single Selection Mode – The default mode. Only one item can be selected at a time.
  - Multiple Interval Selection Mode – Multiple items may be selected.
- You change the selection mode with the control's `getSelectionModel().setSelectionMode()`method.

ListView Controls (12 of 14)

- To change to multiple interval selection mode:

```
listView.getSelectionModel().setSelectionMode(Selec
   tionMode.MULTIPLE);
```

- To change back to single selection mode:

```
listView.getSelectionModel().setSelectionMode(Selec
   tionMode.SINGLE);
```

`ListView` Controls (13 of 14)

- When a `ListView` control is in multiple selection mode, the user can select more than one item.
- To get all of the selected items and their indices, use the following methods:
  - `getSelectionModel().getSelectedItems()` — returns a read-only `ObservableList` of the selected items
  - `getSelectionModel().getSelectedIndices()` — returns a read-only `ObservableList` of the integer indices of the selected items

ListView Controls (14 of 14)

- By default, `ListView` controls are vertically oriented.
- You can set the orientation to either vertical or horizontal with the `ListView` class's `setOrientation` method.
- When you call the method, you pass one of the following enum constants:
  - `Orientation.VERTICAL`
  - `Orientation.HORIZONTAL`
  - (The `Orientation` enum is in the `javafx.geometry` package.)

```
ListView<String> listView = new ListView<>();
listView.getItems().addAll("Monday", "Tuesday", "Wednesday",
                           "Thursday", "Friday", "Saturday",
                           "Sunday");
listView.setOrientation(Orientation.HORIZONTAL);
listView.setPrefSize(200, 50);
```

ComboBox Controls (1 of 7)

- A ComboBox presents a drop-down list of items that the user may select from.
- Use the ComboBox class (in the `javafx.scene.control` package) to create a ComboBox control:

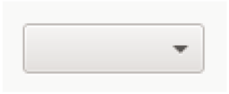
```
ComboBox<String> nameComboBox = new ComboBox<>();
```

- Once you have created a ComboBox control, you are ready to add items to it:

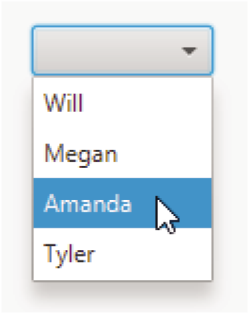
```
nameComboBox.getItems().addAll("Will", "Megan",
   "Amanda", "Tyler");
```

ComboBox Controls (2 of 7)

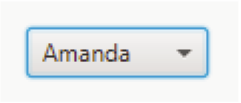
The ComboBox's initial appearance



When the user clicks the ComboBox, a list of items drops down.



The selected item is displayed by the ComboBox.



ComboBox Controls (3 of 7)

Some of the ComboBox Methods

Method	Description
getValue()	Returns the item that is currently selected in the ComboBox.
setValue(*value*)	Selects *value* in the ComboBox.
setVisibleRowCount(*count*)	The *count* argument is an int. Sets the number of rows, or items, to display in the drop-down list.
setEditable(*value*)	The *value* argument is a boolean. If *value* is true, the ComboBox will be editable. If *value* is false, the ComboBox will be uneditable.
show()	Displays the drop-down list of items.
hide()	Hides the drop-down list of items.
isShowing()	Returns true or false to indicate whether the dropdown list is visible.

ComboBox Controls (4 of 7)

- You can use the `ComboBox` class's `getValue()` method get the item that is currently selected:

```
String selected = comboBox.getValue();
```

- If no item is selected in the `ComboBox`, the method will return `null`.
- See `ComboBoxDemo1.java` in your textbook

ComboBox Controls (5 of 7)

- When the user selects an item in a `ListView`, an `ActionEvent` occurs.
- To immediately perform an action when the user selects an item in a ListView, write an event handler that responds to the `ActionEvent`:

```
comboBox.setOnAction(event ->
   {
      // Write event handling code here...
   });
```

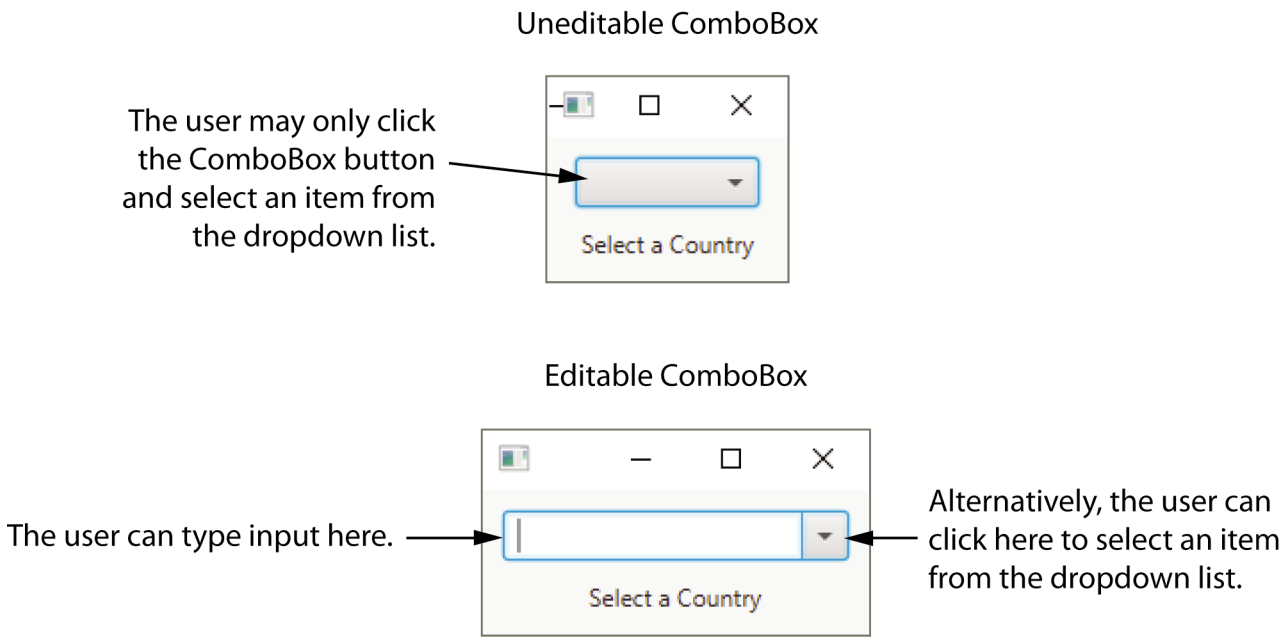
- See ComboBoxDemo2.java in your textbook

ComboBox Controls (6 of 7)

- By default, `ComboBox` controls are uneditable
  - The user cannot type input into the control; he or she can only select items from a drop-down list.
- An editable `ComboBox` allows the user to select an item, *or* type input into a field that is similar to a `TextField`.
- Use the `setEditable` method to make a `ComboBox` editable:

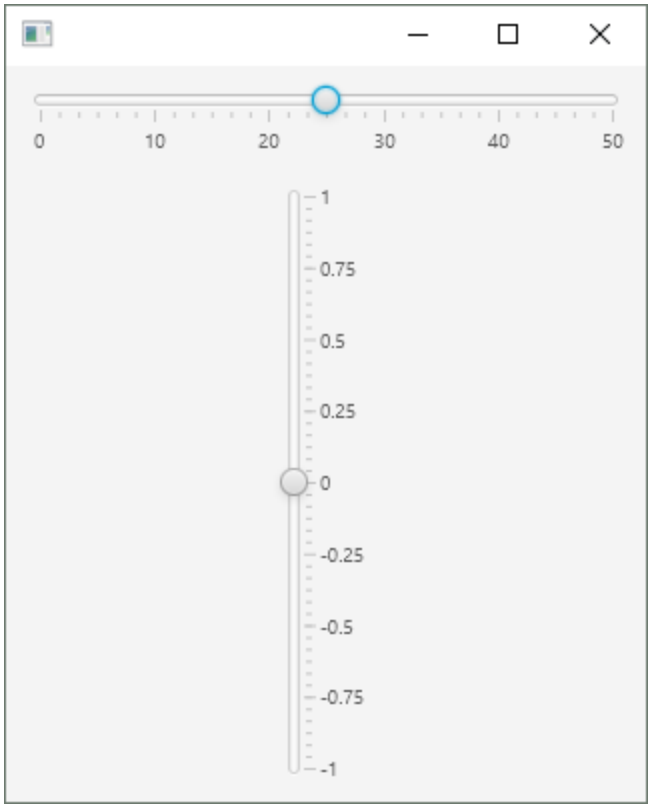
```
ComboBox<String> comboBox = new ComboBox<>();
comboBox.setEditable(true);
comboBox.getItems().addAll("England", "Scotland",
                           "Ireland", "Wales");
```

ComboBox Controls (7 of 7)



`Slider` Controls (1 of 5)

- A `Slider` allows the user to graphically adjust a number within a range.
- `Sliders` are created from the `Slider` class (in the `javafx.scene.control` package)
- They display an image of a "slider knob" that can be dragged along a track.



`Slider` Controls (2 of 5)

- A `Slider` is designed to represent a range of numeric values.
- As the user moves the knob along the track, the numeric value is adjusted accordingly.
- Between the minimum and maximum values, major tick marks are displayed with a label indicating the value at that tick mark.
- Between the major tick marks are minor tick marks.

`Slider` Controls (3 of 5)

- The `Slider` class has two constructors. No-arg:

```
Slider slider = new Slider();
```

- Creates a `Slider` with a minimum value of 0, and a maximum value of 100.

- The `Slider`'s initial value will be set to 0, and no tick marks will be displayed.

`Slider` Controls (4 of 5)

- The second constructor accepts three `double` arguments:
  - the minimum value
  - the maximum value
  - the initial value

```
Slider slider = new Slider(0.0, 50.0, 25.0);
```

  - See Table 12-7 for a list of many of the `Slider` class methods

Slider Controls (5 of 5)

- When a Slider's value changes, a *change event* occurs.
- To perform an action when the Slider's value changes, write an event handler that responds to the change event:

```
slider.valueProperty().addListener((observeable, oldvalue, newvalue) ->
{
   // Write event handling code here...
});
```

- See SliderDemo.java in your textbook

TextArea Controls (1 of 4)

- A `TextArea` is a multiline `TextField` that can accept several lines of text input, or display several lines of text.

- The `TextArea` class (in the `javafx.scene.control` package) has two constructors.

```
TextArea textArea = new TextArea();

TextArea textArea = new TextArea("This is the initial text.");
```

TextArea Controls (2 of 4)

Some of the TextArea Class Methods

Method	Description
setPrefColumnCount(*value*)	The *value* argument is an int. Sets the preferred number of text columns. (The default value is 40.)
setPrefRowCount(*value*)	The *value* argument is an int. Sets the preferred number of text rows. (The default value is 10.)
setWrapText(*value*)	The *value* argument is a boolean. If the argument is true, then the TextArea will wrap a line of text that exceeds the end of a row. If the argument is false, then the TextArea will scroll when a line of text exceeds the end of a row. (The default value is false.)
getText()	Returns the contents of the TextArea as a String.
setText(*value*)	The *value* argument is a String. The argument becomes the text that is displayed in the TextArea control.

TextArea Controls (3 of 4)

- Example: Creating an empty TextArea control with a width of 20 columns and a height of 30 rows:

```
TextArea textArea = new TextArea();
textArea.setPrefColumnCount(20);
textArea.setPrefRowCount(30);
```

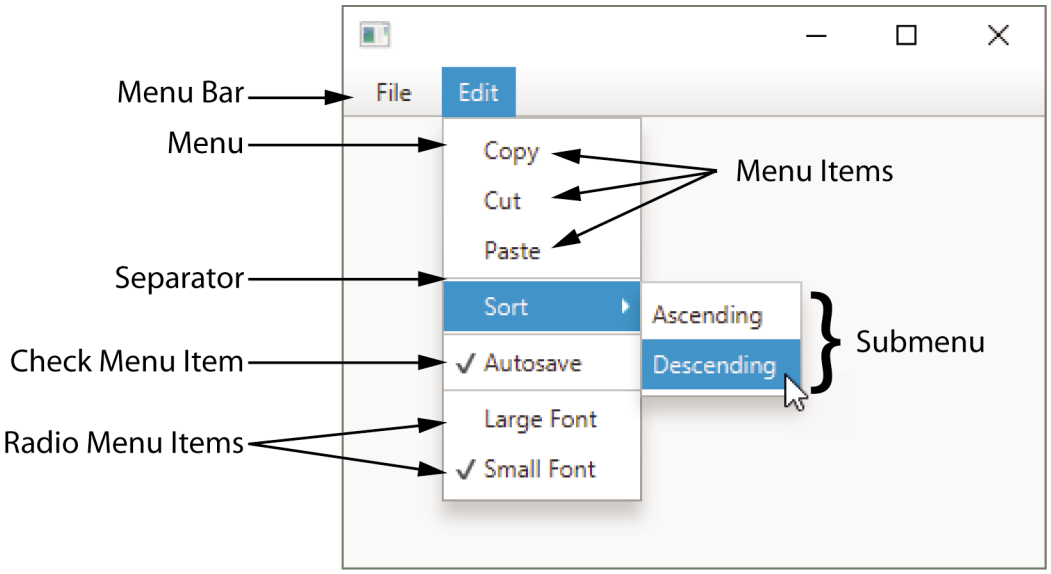
TextArea Controls (4 of 4)

- By default, `TextArea` controls do not perform text wrapping.
- To enable text wrapping, use the `setWrapText` method.
- The method accepts a `boolean` argument: `true` = text wrapping is enabled, `false` = text wrapping is disabled.

```
textInput.setWrapText(true);
```

Menus (1 of 5)

- A menu system is a collection of commands organized in one or more drop-down menus.



Menus (2 of 5)

- A menu system commonly consists of:
  - Menu Bar – lists the names of one or more menus
  - Menu – a drop-down list of menu items
  - Menu Item – can be selected by the user
  - Check box menu item – appears with a small box beside it
  - Radio button menu item – may be selected or deselected
  - Submenu – a menu within a menu
  - Separator bar – used to separate groups of items

Menus (3 of 5)

- A menu system is constructed with the following classes:
  - MenuBar – used to create a menu bar
  - Menu – used to create a menu, containing:
    - MenuItem
    - CheckMenuItem
    - RadioMenuItem
    - other Menu objects
  - MenuItem – Used to create a regular menu, and generates an ActionEventwhen selected.

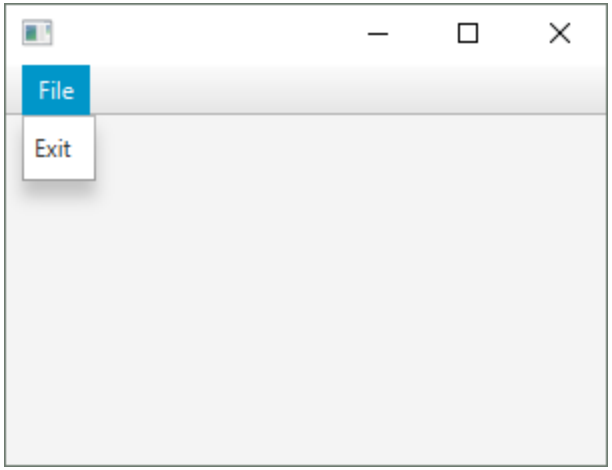
Menus (4 of 5)

- `CheckMenuItem` – Used to create a checked menu item.
  - The class's `isSelected` method returns `true` if the item is selected, or `false` otherwise.
  - A `CheckMenuItem` component generates an `ActionEvent` when selected.

- `RadioMenuItem` – Used to create a radio button menu item.
  - `RadioMenuItem` components can be grouped together in a `ToggleGroup` so that only one of them can be selected at a time.
  - The class's `isSelected` method returns `true` if the item is selected, or `false` otherwise.
  - A `RadioMenuItem` component generates an `ActionEvent` when selected

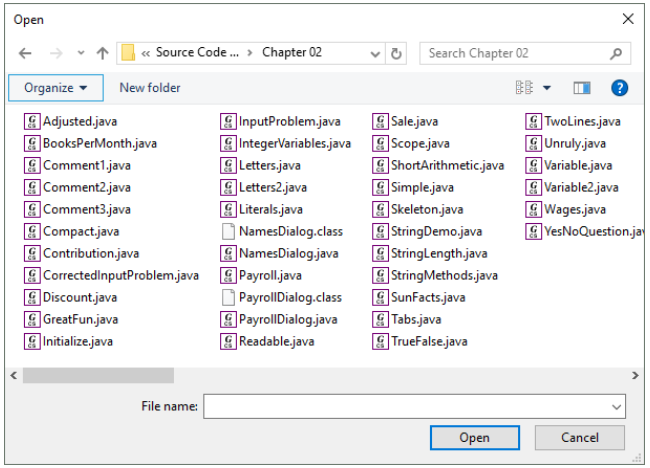
Menus (5 of 5)

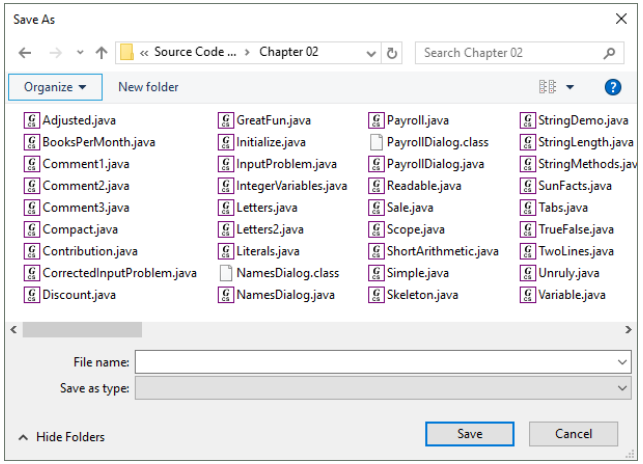
- See `SimpleMenu.java` in your textbook for an example



The `FileChooser` Class (1 of 3)

- The `FileChooser` class (in the `javafx.stage` package) displays a dialog box that allows the user to browse for a file and select it.
- The class can display two types of predefined dialog boxes:
  - open dialog box
  - save dialog box





The `FileChooser` Class (2 of 3)

- First, create an instance of the `FileChooser` class
- Then, call either the `showOpenDialog` method, or the `showSaveDialog` method
- With either method, you pass a reference to the application's stage as an argument

```
FileChooser fileChooser = new FileChooser();
File selectedFile = fileChooser.showOpenDialog(primaryStage);
```

The `FileChooser` Class (3 of 3)

- The `showOpenDialog` and `showSaveDialog` methods return a `File` object (in the `java.io` package) containing information about the selected file.
- If the user does not select a file, the method returns `null`.

```
FileChooser fileChooser = new FileChooser();
File selectedFile = fileChooser.showOpenDialog(primaryStage);
if (selectedFile != null)
{
   String filename = selectedFile.getPath();
   outputLabel.setText("You selected " + filename);
}
```

Using Console Output to Debug a GUI

- Use `System.out.println()` to display variable values, etc. in the console as your application executes to identify logic errors

```
// For debugging, display the text entered, and
// its value converted to a double.
System.out.println("Reading " + str +
                   " from the text field.");
System.out.println("Converted value: " +
                   Double.parseDouble(str));
```

- See `ConsoleDebugging.java` in your textbook

Copyright