CHAPTER 11

ADVANCED USER INTERFACES (FX)

© Carlos Santa Maria/iStockphoto.

CHAPTER GOALS

To arrange user-interface controls

To become familiar with common user-interface controls, such as radio buttons, check boxes, and menus

To understand how to update user interfaces with properties and bindings

To build programs that show animations and handle mouse events

CHAPTER CONTENTS

© Carlos Santa Maria/iStockphoto.

The graphical applications with which you are familiar have many visual gadgets for information entry: buttons, scroll bars, menus, and so on. In this chapter, you will learn how to use the most common user-interface components in the JavaFX toolkit. You will also learn more about event handling, so you can use timer events in animations and process mouse events in interactive graphical programs.

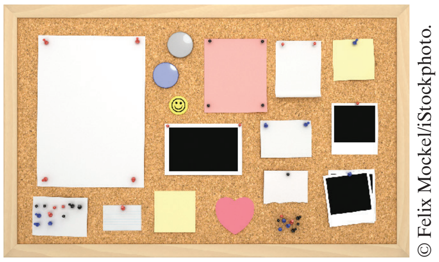
11.1 Layout Management

Up to now, you have had limited control over the layout of user-interface components. You learned how to add controls to a pane and relocate them, and also how to add them to a VBox pane that arranges objects from top to bottom. However, in many applications, you need more sophisticated arrangements.

11.1.1 Horizontal and Vertical Boxes

In JavaFX, you use layout panes to arrange user-interface controls.

In JavaFX, you build up user interfaces by adding controls into **layout panes**. You have already seen the VBox that arranges its children vertically. There is an analogous HBox for horizontal arrangement.

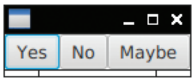

© Felix Mockel/iStockphoto.

A layout pane arranges user-interface components.

When we used the VBox pane in the preceding chapter, you may have noticed that there was no gap between the controls, and none between the controls and the window holding them. The same is true for the HBox pane. Consider this code:

```
Button button1 = new Button("Yes");
Button button2 = new Button("No");
Button button3 = new Button("Maybe");
HBox buttons = new HBox(button1, button2, button3);
Scene scene1 = new Scene(buttons);
stage1.setScene(scene1);
stage1.show();
```

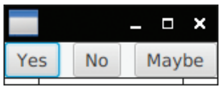
The result looks like this:



That is not very attractive. First, set a gap *between* controls in the HBox constructor:

```
HBox buttons = new HBox(10, button1, button2, button3);
```

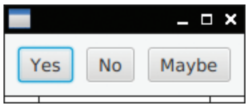
Now the buttons are separated by ten pixels.



To add space *around* the controls, set the *padding* like this:

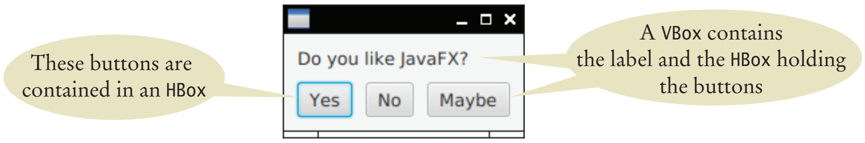
```
buttons.setPadding(new Insets(10));
```

With these settings, the appearance is much improved:



You can arrange controls by placing them in nested HBox and VBox panes.

You can nest HBox and VBox panes to produce more complex layouts. For example, here we have a VBox with a label and an HBox holding three buttons:


Do you like JavaFX?
Yes
No
Maybe
These buttons are contained in an HBox
A VBox contains the label and the HBox holding the buttons

This layout is achieved with the following code:

```
Label question = new Label("Do you like JavaFX?");
Button button1 = new Button("Yes");
Button button2 = new Button("No");
Button button3 = new Button("Maybe");
HBox buttons = new HBox(10, button1, button2, button3);
VBox root = new VBox(10, question, buttons);
root.setPadding(new Insets(10));
Scene scene1 = new Scene(root);
```

Nesting horizontal panels inside vertical panels works fine as long as you are not concerned about alignment between columns. The next section shows you how to realize more complex layouts. Here is the code for creating the nested boxes.

sec01_01/BoxDemo.java

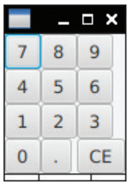
```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Pane;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class BoxDemo extends Application
{
   public void start(Stage stage1)
   {
      Pane root = createRootPane();
      Scene scene1 = new Scene(root);
      stage1.setScene(scene1);
      stage1.setTitle(" ");
      stage1.show();
   }

   public Pane createRootPane()
   {
      Button button1 = new Button("Yes");
      Button button2 = new Button("No");
```

```
        Button button3 = new Button("Maybe");
        Pane buttons = new HBox(10, button1, button2, button3);
        Label question = new Label("Do you like JavaFX?");

        VBox root = new VBox(10, question, buttons);

        root.setPadding(new Insets(10));
        return root;
    }
}
```

11.1.2 The Grid Pane

Use a GridPane to lay out controls in a grid of rows and columns.

The GridPane arranges controls in a grid of rows and columns. A good example is a calculator keypad, such as this:



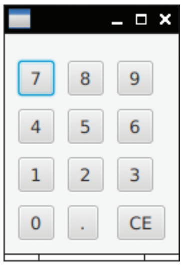
When you add a control to a GridPane, you specify the column and row position. (The column position comes first because it is the *x*-position in a mathematical coordinate system, which comes traditionally before the *y*-position that indicates the row.)

```
GridPane pane = new GridPane();
pane.add(new Button("7"), 0, 0);
pane.add(new Button("8"), 1, 0);
pane.add(new Button("9"), 2, 0);
pane.add(new Button("4"), 0, 1);
. . .
```

To add some space between the rows and columns, and around the edges, use these statements:

```
pane.setHgap(10);
pane.setVgap(10);
pane.setPadding(new Insets(10));
```

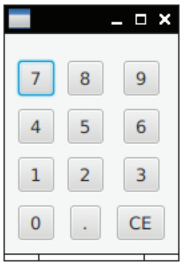
Here is the result:



If you want to center a component within its column, you have to work a bit harder:

```
Button button7 = new Button("7");
pane.add(button7, 0, 0);
GridPane.setHalignment(button7, HPos.CENTER);
```

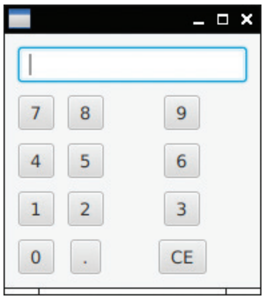
Here is how the calculator looks when all buttons have been centered:



A pocket calculator also has a display that shows the value of the last calculation. That display spans all columns. When you add a control to a grid pane that should span multiple columns or rows, you indicate those column and row counts in the `add` method:

```
TextField display = new TextField();
pane.add(display, 0, 0, 3, 1); // Spans 3 columns and one row
```

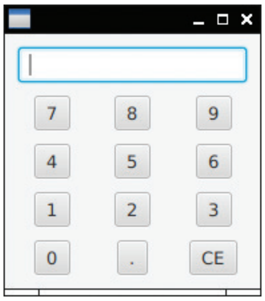
The text field now spans three columns:



By default, the columns don't have equal size. Instead, each column is as narrow as possible to hold its children, and the last column receives any remaining space. You can adjust the column widths like this:

```
ColumnConstraints col1 = new ColumnConstraints();
col1.setPercentWidth(33.33);
ColumnConstraints col2 = new ColumnConstraints();
col2.setPercentWidth(33.33);
ColumnConstraints col3 = new ColumnConstraints();
col3.setPercentWidth(33.33);
pane.getColumnConstraints().addAll(col1,col2,col3);
```

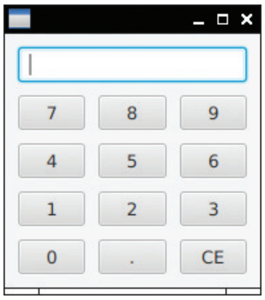
With these statements, all columns have the same size:



If you look carefully at the buttons, you will note that their sizes differ slightly. By default, buttons don't grow beyond their preferred size. You can fix that by setting their maximum width. If you want the button to grow as much as possible to fill a column, you can set the maximum width to a very large value, such as 1,000, or even better, to the largest possible floating-point value:

```
button7.setMaxWidth(Double.MAX_VALUE);
```

If you do that with all buttons, each of them grows to fill the column. Now we have a calculator with a nice layout of its child components:



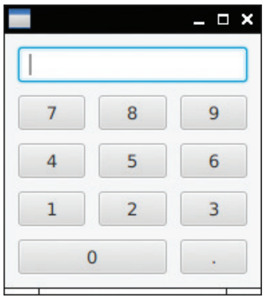
Here is the complete program. To keep the code from being repetitive, the buttons are set up in a loop. This program just shows how to achieve the layout. Worked Example 11.1 makes the buttons active.

sec01_02/Calculator.java

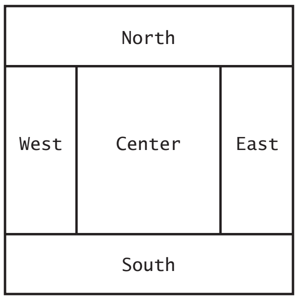
```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.TextField;
import javafx.scene.layout.ColumnConstraints;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;

public class Calculator extends Application
{
   public void start(Stage stage1)
   {
      Pane root = createRootPane();
      Scene scene1 = new Scene(root);
      stage1.setScene(scene1);
      stage1.setTitle(" ");
      stage1.show();
   }

   public Pane createRootPane()
   {
      GridPane pane = new GridPane();
      pane.setHgap(10);
      pane.setVgap(10);
      pane.setPadding(new Insets(10));

      TextField display = new TextField("");
      pane.add(display, 0, 0, 3, 1);

      String[] labels =
            { "7", "8", "9", "4", "5", "6", "1", "2", "3", "0", ".", "CE" };
      int r = 1;
      int c = 0;
      for (String label : labels)
      {
         Button b = new Button(label);
         b.setMaxWidth(Double.MAX_VALUE);
         pane.add(b, c, r);
         c++;
         if (c == 3) { c = 0; r++; }
      }
      ColumnConstraints col1 = new ColumnConstraints();
      col1.setPercentWidth(33.33);
      ColumnConstraints col2 = new ColumnConstraints();
      col2.setPercentWidth(33.33);
      ColumnConstraints col3 = new ColumnConstraints();
      col3.setPercentWidth(33.33);
      pane.getColumnConstraints().addAll(col1, col2, col3);
      return pane;
   }
}
```

SELF CHECK

1. How can you arrange the "Yes", "No", and "Maybe" buttons vertically instead of horizontally?
2. How can you create a calculator using only `HBox` and `VBox` panes?
3. What happens if you place two buttons in the same position of a grid pane? Try it out with a small program.
4. Some calculators have a double-wide `0` button, as shown below. How can you achieve that?



5. The `BorderPane` arranges five components in the following configuration:



How can you achieve the same effect with a `GridPane`?

Practice It Now you can try these exercises at the end of the chapter: R11.3, E11.1, E11.2.

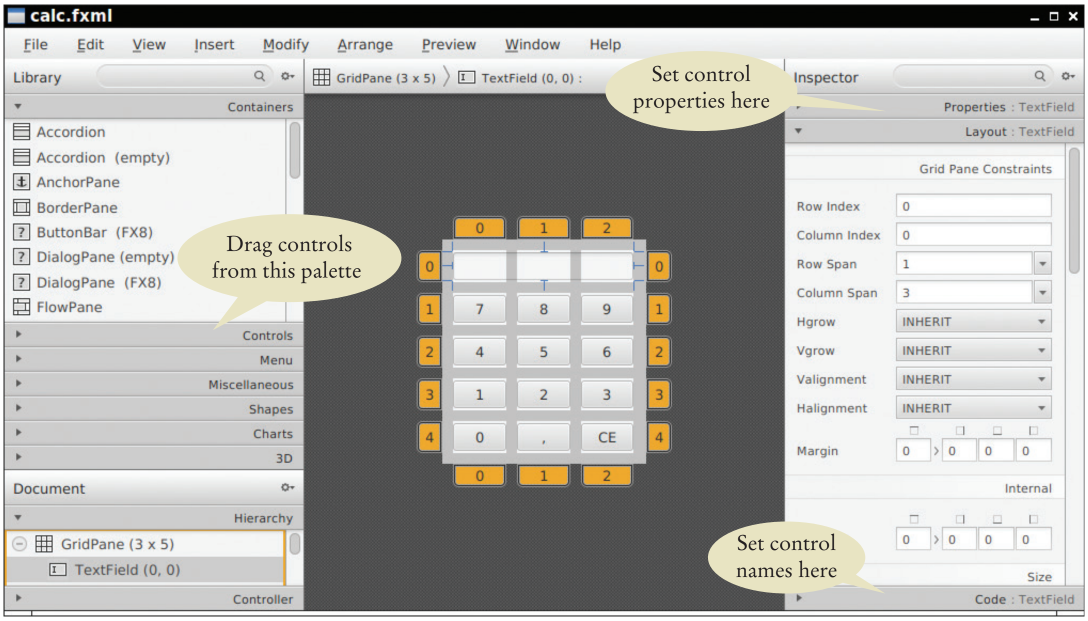
Programming Tip 11.1

Use a GUI Builder

As you have seen, implementing even a simple graphical user interface in Java is quite tedious. You have to write a lot of code for using layout panes and fine-tuning the layout of controls.

A GUI builder takes away much of the tedium. You simply drag and drop controls, and pick their layout properties from a set of choices. (To understand the offered choices, it helps to know how to write the user interface programmatically.)

The standard GUI builder for JavaFX is called Scene Builder. You can download it from `http://gluonhq.com/labs/scene-builder/`. To produce the layout for the calculator, drag a grid pane to the center area. Right-click and select the options to add rows and columns. Then drag a text field and the buttons to their desired locations, and set their properties (see Figure 1).



Row Index 0
Column Index 0
Row Span 1
Column Span 3
Hgrow INHERIT
Vgrow INHERIT
Valignment INHERIT
Halignment INHERIT
Margin 0 0 0 0

Figure 1 The JavaFX Scene Builder

When you are satisfied with the layout, save your work as a file `calc.fxml` and place it into the same directory as the Java classes of your program. You can peek inside the file—it contains your layout instructions in a format called FXML.

Then you can load the layout with the following code:

```
public class Calculator extends Application
{
   public void start(Stage stage1)
   {
      Parent root = null;
      try
      {
         root = FXMLLoader.load(getClass().getResource("calc.fxml"));
         Scene scene1 = new Scene(root);
```

```
            stage1.setScene(scene1);
            stage1.setTitle(" ");
            stage1.show();
         }
         catch (IOException e)
         {
            System.out.println("Couldn't load calc.fxml");
         }
      }
   }
```

Now the user interface is displayed, just as with the program of the preceding section. Note that you do not have to program any layout commands.

However, you still need to attach handlers to the buttons. Therefore, you need to be able to access them in your program. First, provide a *controller* class that attaches the handlers. Provide an instance variable for each control that you'd like to access from your controller class, and annotate it with the annotation `@FXML`:

```
public class CalculatorController implements Initializable
{
   @FXML private TextField display;
   @FXML private Button button0;
   @FXML private Button button1;
   . . .
   public void initialize(URL url, ResourceBundle rb)
   {
      . . .
   }
}
```

In the `initialize` method, set the event handlers;

```
public void initialize(URL url, ResourceBundle rb)
{
   button0.setOnAction(event -> display.appendText("0"));
   button1.setOnAction(event -> display.appendText("1"));
   . . .
}
```

In Scene Builder, you need to set the `fx:id` attribute of every control to the same name as the instance variable that is annotated with `@FXML`. You find the setting in the "Code" pane in the right-hand side of the Scene Builder program. You also need to set the controller class in the "Controller" pane in the bottom-left corner.

When the FXML file is loaded, the annotated instance variables are initialized with the controls. If there is a problem with the initialization, look into the FXML and check the `fx:id` and `fx:controller` attributes.

By using FXML, the layout and code are separated. A professional user interface designer, who need not be a programmer, can provide attractive and functional user interfaces.

programming_tip_1/CalculatorController.java

```
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Button;
import javafx.scene.control.TextField;

import java.net.URL;
import java.util.ResourceBundle;

public class CalculatorController implements Initializable
{
   @FXML private TextField display;
   @FXML private Button button0;
   @FXML private Button button1;
   @FXML private Button button2;
   @FXML private Button button3;
   @FXML private Button button4;
   @FXML private Button button5;
   @FXML private Button button6;
   @FXML private Button button7;
   @FXML private Button button8;
   @FXML private Button button9;
   @FXML private Button buttonDP;
   @FXML private Button buttonCE;

   public void initialize(URL url, ResourceBundle rb)
   {
      button0.setOnAction(event -> display.appendText("0"));
      button1.setOnAction(event -> display.appendText("1"));
      button2.setOnAction(event -> display.appendText("2"));
      button3.setOnAction(event -> display.appendText("3"));
      button4.setOnAction(event -> display.appendText("4"));
      button5.setOnAction(event -> display.appendText("5"));
      button6.setOnAction(event -> display.appendText("6"));
      button7.setOnAction(event -> display.appendText("7"));
      button8.setOnAction(event -> display.appendText("8"));
      button9.setOnAction(event -> display.appendText("9"));
      buttonDP.setOnAction(event -> display.appendText("."));
      buttonCE.setOnAction(event -> display.setText(""));
   }
}
```

programming_tip_1/Calculator.java

```
import java.io.IOException;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Calculator extends Application
{
   public void start(Stage stage1)
   {
      Parent root = null;
      try
      {
         root = FXMLLoader.load(getClass().getResource("calc.fxml"));
         Scene scene1 = new Scene(root);
         stage1.setScene(scene1);
         stage1.setTitle(" ");
         stage1.show();
      }
      catch (IOException e)
      {
         System.out.println("Couldn't load calc.fxml");
      }
```

```
    }
}
```

programming_tip_1/calc.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.Insets?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.ColumnConstraints?>
<?import javafx.scene.layout.GridPane?>

<GridPane hgap="10.0" vgap="10.0" xmlns="http://javafx.com/javafx/8.0.102"
      xmlns:fx="http://javafx.com/fxml/1" fx:controller="CalculatorController">
  <columnConstraints>
    <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" percentWidth="33.0" />
    <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" percentWidth="33.0" />
    <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" percentWidth="33.0" />
  </columnConstraints>
  <children>
    <TextField fx:id="display" GridPane.columnSpan="3" />
    <Button fx:id="button7" maxWidth="1.7976931348623157E308" mnemonicParsing="false" text="7"
       GridPane.rowIndex="1" />
    <Button fx:id="button8" maxWidth="1.7976931348623157E308" mnemonicParsing="false" text="8"
       GridPane.columnIndex="1" GridPane.rowIndex="1" />
    <Button fx:id="button4" maxWidth="1.7976931348623157E308" mnemonicParsing="false" text="4"
       GridPane.rowIndex="2" />
    <Button fx:id="button5" maxWidth="1.7976931348623157E308" mnemonicParsing="false" text="5"
       GridPane.columnIndex="1" GridPane.rowIndex="2" />
    <Button fx:id="button9" maxWidth="1.7976931348623157E308" mnemonicParsing="false" text="9"
       GridPane.columnIndex="2" GridPane.rowIndex="1" />
    <Button fx:id="button6" maxWidth="1.7976931348623157E308" mnemonicParsing="false" text="6"
       GridPane.columnIndex="2" GridPane.rowIndex="2" />
    <Button fx:id="button1" maxWidth="1.7976931348623157E308" mnemonicParsing="false" text="1"
       GridPane.rowIndex="3" />
    <Button fx:id="button2" maxWidth="1.7976931348623157E308" mnemonicParsing="false" text="2"
       GridPane.columnIndex="1" GridPane.rowIndex="3" />
    <Button fx:id="button3" maxWidth="1.7976931348623157E308" mnemonicParsing="false" text="3"
       GridPane.columnIndex="2" GridPane.rowIndex="3" />
    <Button fx:id="button0" maxWidth="1.7976931348623157E308" mnemonicParsing="false" text="0"
       GridPane.rowIndex="4" />
    <Button fx:id="buttonDP" maxWidth="1.7976931348623157E308" mnemonicParsing="false" text="."
       GridPane.columnIndex="1" GridPane.rowIndex="4" />
    <Button fx:id="buttonCE" maxWidth="1.7976931348623157E308" mnemonicParsing="false" text="CE"
       GridPane.columnIndex="2" GridPane.rowIndex="4" />
  </children>
  <padding>
    <Insets bottom="10.0" left="10.0" right="10.0" top="10.0" />
  </padding>
</GridPane>
```

Special Topic 11.1

Styling with CSS

Instead of programming user interface details such as

```
pane.setHgap(10);
pane.setVgap(10);
pane.setPadding(new Insets(10));
```

you can use cascading style sheets (CSS), a technology for describing the appearance of web pages. JavaFX uses an adaptation of the CSS standard.

Put formatting instructions into a separate file, using the CSS syntax:

```
GridPane {
   -fx-hgap: 0.5em;
   -fx-vgap: 0.5em;
   -fx-padding: 0.5em;
}

GridPane Button {
   -fx-max-width: 100em;
}
```

We won't cover the CSS syntax here, but it is pretty easy to understand. These statements indicate how to set the gaps and padding of a `GridPane`, and the maximum width of all buttons inside a grid pane. The "em" measurement means "the width of a lowercase letter m." Using em is better than using pixels because some users have very high resolution displays, in which each individual pixel is tiny.

If you want to provide a style for a specific control (not all controls of a given type), set the control's ID in the JavaFX code:

```
button.setId("buttonCE");
```

Then you can reference the specific control in the style sheet like this:

```
#buttonCE {
    -fx-background-color: lightpink;
}
```

To use the style sheet, add it to the scene:

```
scene1.getStylesheets().add("calc.css");
```

If you like, you can use an FXML file and a style sheet. In Scene Builder, select the root pane, then the Properties tab, and add the style sheet. Use the `id` property of individual controls to assign CSS IDs to them.

You can also apply CSS styles without a style sheet. Sometimes, it is easier to create an effect with CSS than with JavaFX features. For example, this command adds a dotted blue border around a pane:

```
pane.setStyle("-fx-border-style: dotted;"
   + " -fx-border-width: 1px;"
   + " -fx-border-color: blue;");
```

That is easier than using the `Border` class:

```
pane.setBorder(
   new Border(
      new BorderStroke(Color.BLUE,
         BorderStrokeStyle.DOTTED,
         CornerRadii.EMPTY,
         BorderStroke.THIN)));
```

Here is the code for a calculator styled with CSS.

special_topic_1/Calculator.java

```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.TextField;
import javafx.scene.layout.ColumnConstraints;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;

public class Calculator extends Application
{
   public void start(Stage stage1)
   {
      Pane root = createRootPane();
      Scene scene1 = new Scene(root);
      scene1.getStylesheets().add("calc.css");
      stage1.setScene(scene1);
      stage1.setTitle(" ");
      stage1.show();
   }

   public Pane createRootPane()
   {
      GridPane pane = new GridPane();

      TextField display = new TextField("");
      pane.add(display, 0, 0, 3, 1);

      String[] labels = { "7", "8", "9", "4", "5", "6", "1", "2", "3",
         "0", ".", "CE" };
      int r = 1;
      int c = 0;
      for (String label : labels)
      {
         Button b = new Button(label);
         pane.add(b, c, r);
         c++;
         if (c == 3) { c = 0; r++; }
         b.setId("button" + label);
      }
      // This can't be done with CSS
      ColumnConstraints col1 = new ColumnConstraints();
      col1.setPercentWidth(33.33);
      ColumnConstraints col2 = new ColumnConstraints();
      col2.setPercentWidth(33.33);
      ColumnConstraints col3 = new ColumnConstraints();
      col3.setPercentWidth(33.33);
      pane.getColumnConstraints().addAll(col1, col2, col3);
      return pane;
   }
}
```

special_topic_1/calc.css

```css
GridPane {
    -fx-hgap: 0.5em;
    -fx-vgap: 0.5em;
    -fx-padding: 0.5em;
}

GridPane Button {
    -fx-max-width: 100em;
}

#buttonCE {
    -fx-background-color: lightpink;
}
```

11.2 Choices

In the following sections, you will see how to present a finite set of choices to the user. Which FX control you use depends on whether the choices are mutually exclusive or not, and on the amount of space you have for displaying the choices.

11.2.1 Radio Buttons

For a small set of mutually exclusive choices, use a group of radio buttons or a choice box.

If the choices are *mutually exclusive*, use a set of **radio buttons**. In a radio button set, only one button can be selected at a time. When the user selects another button in the same set, the previously selected button is automatically turned off. (These buttons are called radio buttons because they work like the station selector buttons on a car radio: If you select a new station, the old station is automatically deselected.) For example, in Figure 2, the font sizes are mutually exclusive. You can select small, medium, or large, but not a combination of them.

© Michele Cornelius/iStockphoto.

In an old fashioned radio, pushing down one station button released the others.

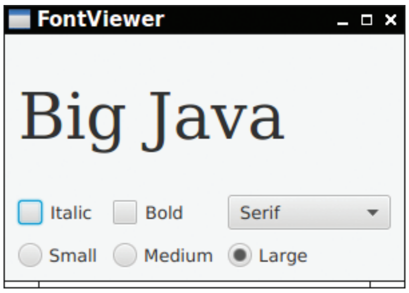


Figure 2 Check Boxes, a Choice Box, and Radio Buttons

To create a set of radio buttons, create each button individually, and add all buttons in the set to a ToggleGroup object:

```
RadioButton smallButton = new RadioButton("Small");
RadioButton mediumButton = new RadioButton("Medium");
RadioButton largeButton = new RadioButton("Large");
```

```
ToggleGroup group = new ToggleGroup();
smallButton.setToggleGroup(group);
mediumButton.setToggleGroup(group);
largeButton.setToggleGroup(group);
```

Add radio buttons to a ToggleGroup so that only one button in the group is selected at any time.

Note that the toggle group does not place the buttons close to each other in the container. The purpose of the toggle group is simply to find out which buttons to turn off when one of them is turned on. It is still your job to arrange the buttons on the screen.

The `isSelected` method is called to find out whether a button is currently selected or not. For example,

```
if (largeButton.isSelected()) { size = LARGE_SIZE; }
```

Because users will expect one radio button in a radio button group to be selected, you should call `setSelected(true)` on one of the radio buttons when you set up the user interface.

11.2.2 Check Boxes

For a binary choice, use a check box.

A **check box** is a user-interface component with two states: checked and unchecked. You use a group of check boxes when one selection does not exclude another. For example, the choices for "Bold" and "Italic" in Figure 2 are not exclusive. You can choose either, both, or neither. Therefore, they are implemented as a set of separate check boxes. Radio buttons and check boxes have different visual appearances. Radio buttons are round and have a black dot when selected. Check boxes are square and have a check mark when selected.

You construct a check box by providing the name in the constructor:

```
CheckBox italicCheckBox = new CheckBox("Italic");
```

Because check box settings do not exclude each other, you do not place a set of check boxes inside a toggle group.

As with radio buttons, you use the `isSelected` method to find out whether a check box is currently checked or not.

11.2.3 Choice Boxes

For a large set of choices, use a choice box.

If you have a large number of choices, you don't want to make a set of radio buttons, because that would take up a lot of space. Instead, you can use a *choice box*. When you click on the arrow icon to the right of the text field of a choice box, a list of selections drops down, and you can choose one of the items in the list (see Figure 3).

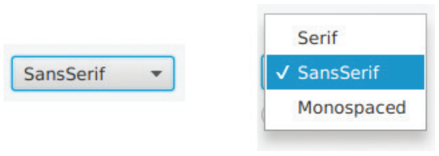


Figure 3 Opening a Choice Box

Add strings to a choice box like this:

```
ChoiceBox<String> fontChoice = new ChoiceBox<>();
fontChoice.getItems().addAll("Serif", "SansSerif", "Monospaced");
```

You get the item that the user has selected by calling the `getSelectedItem` method on the object that the `getSelectionModel` method returns.

```
String facename = fontChoice.getSelectionModel().getSelectedItem();
```

You can select an item for the user by invoking the `select` method on the object returned by the `getSelectionModel` method:

```
fontChoice.getSelectionModel().select("Serif");
```

Radio buttons, check boxes, and choice boxes generate action events, just as buttons do.

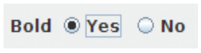
Radio buttons, check boxes, and choice boxes generate an `ActionEvent` whenever the user selects an item. In the following program, we don't care which component was clicked—all components notify the same listener object. Whenever the user clicks on any one of them, we simply ask each control for its current content. We then update the font of the label.

sec02/FontViewer.java

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.control.Label;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Pane;
import javafx.scene.text.Font;
import javafx.scene.text.FontPosture;
import javafx.scene.text.FontWeight;
import javafx.stage.Stage;

public class FontViewer extends Application
{
   private Label sample;
   private CheckBox italicCheckbox;
   private CheckBox boldCheckbox;
   private RadioButton smallButton;
   private RadioButton mediumButton;
   private RadioButton largeButton;
   private ChoiceBox<String> fontChoice;

   public void start(Stage primaryStage)
   {
      Pane root = createRootPane();
      Scene scene1 = new Scene(root);
      primaryStage.setScene(scene1);
      primaryStage.setTitle("FontViewer");
      primaryStage.show();
   }

   private Pane createRootPane()
   {
      sample = new Label("Big Java");
      italicCheckbox = new CheckBox("Italic");
      italicCheckbox.setOnAction(event -> updateSample());

      boldCheckbox = new CheckBox("Bold");
      boldCheckbox.setOnAction(event -> updateSample());

      ToggleGroup group = new ToggleGroup();
      smallButton = new RadioButton("Small");
      smallButton.setToggleGroup(group);
      smallButton.setOnAction(event -> updateSample());
      mediumButton = new RadioButton("Medium");
      mediumButton.setToggleGroup(group);
      mediumButton.setOnAction(event -> updateSample());
      largeButton = new RadioButton("Large");
      largeButton.setToggleGroup(group);
      largeButton.setOnAction(event -> updateSample());
      largeButton.setSelected(true);
      fontChoice = new ChoiceBox<>();
      fontChoice.getItems().addAll("Serif", "SansSerif", "Monospaced");
      fontChoice.getSelectionModel().select("Serif");
      fontChoice.setOnAction(event -> updateSample());

      GridPane pane = new GridPane();

      pane.add(sample, 0, 0, 3, 1);
      pane.add(italicCheckbox, 0, 1);
      pane.add(boldCheckbox, 1, 1);
      pane.add(smallButton, 0, 2);
      pane.add(mediumButton, 1, 2);
      pane.add(largeButton, 2, 2);
      pane.add(fontChoice, 2, 1);

      pane.setHgap(10);
      pane.setVgap(10);

      pane.setPadding(new Insets(10, 10, 10, 10));
      sample.setMinHeight(100);

      updateSample();
      return pane;
   }

   private void updateSample()
   {
      String facename = fontChoice.getSelectionModel().getSelectedItem();
      FontPosture posture;
      if (italicCheckbox.isSelected())
      {
         posture = FontPosture.ITALIC;
      }
      else
      {
         posture = FontPosture.REGULAR;
      }
      FontWeight weight;
      if (boldCheckbox.isSelected())
      {
         weight = FontWeight.BOLD;
      }
      else
      {
         weight = FontWeight.NORMAL;
      }
      // Get font size

      int size = 0;
      final int SMALL_SIZE = 24;
      final int MEDIUM_SIZE = 36;
      final int LARGE_SIZE = 48;

      if (smallButton.isSelected()) { size = SMALL_SIZE; }
      else if (mediumButton.isSelected()) { size = MEDIUM_SIZE; }
      else if (largeButton.isSelected()) { size = LARGE_SIZE; }

      // Set font of label

      sample.setFont(Font.font(facename, weight, posture, size));
   }
}
```

SELF CHECK

6. What is the advantage of a `ChoiceBox` over a set of radio buttons? What is the disadvantage?
7. What happens when you put two check boxes into a toggle group? Try it out if you are not sure.
8. How could the following user interface be improved?



9. Why do all user-interface controls in the `FontViewer` class share the same listener?
10. The static method `Font.getFamilies` yields a `List<String>` with the names of all font families on the user's computer. How should you modify the `FontViewer` program so that the user can choose among all of them?

Practice It Now you can try these exercises at the end of the chapter: E11.4, E11.5, E11.6.

Programming Tip 11.2

Use Chart Controls

Worked Example 10.1 showed how to write a program that creates bar charts. That program is useful for learning about user-interface programming, but if you want to draw a chart, you should use one of the chart controls that comes with JavaFX.

JavaFX has controls for bar charts, pie charts, line charts, and several other chart types. The results look attractive and can be customized in many ways. We don't want to go into the chart classes in detail. To give you a flavor, here are instructions for making a line chart (see Figure 4). You define the *x*- and *y*-axis and then create a chart object:

```
NumberAxis xAxis = new NumberAxis();
xAxis.setLabel("Period");
NumberAxis yAxis = new NumberAxis();
yAxis.setLabel("Balance");
LineChart<Number, Number> chart = new LineChart<>(xAxis, yAxis);
```

Now you assemble a series of data points:

```
XYChart.Series<Number, Number> balances = new XYChart.Series<>();
balances.setName("5%");
for (int i = 0; i <= PERIODS; i++)
{
   balances.getData().add(new XYChart.Data<>(i, balance));
   Update balance.
}
```

Add the series to the chart:

```
chart.getData().add(balances);
```

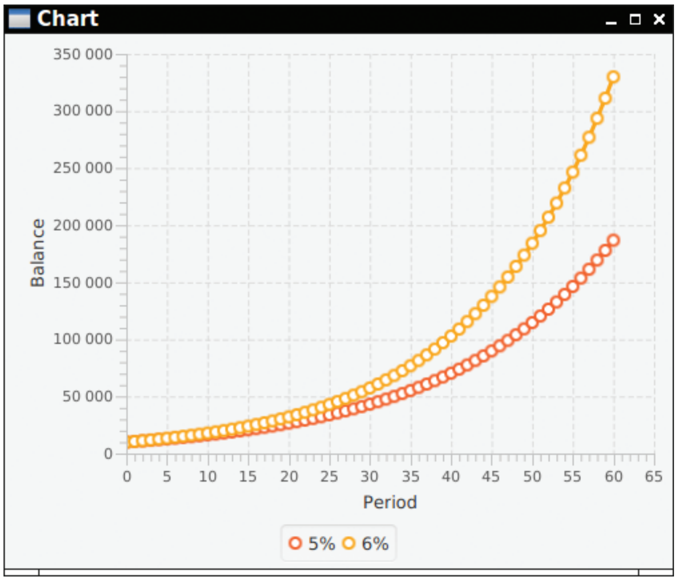


Figure 4 A JavaFX Line Chart

You can add as many series as you like. Figure 4 has one series for compound interest at 5 percent and another for 6 percent.

If you look carefully, you can see that the data points in each series are joined by lines. If you don't want the lines, use a `ScatterChart` instead.

programming_tip_2/Chart.java

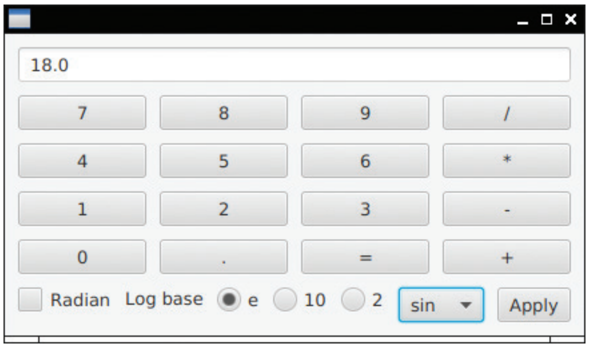
```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.chart.LineChart;
import javafx.scene.chart.NumberAxis;
import javafx.scene.chart.XYChart;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;

public class Chart extends Application
{
   public void start(Stage primaryStage)
   {
      Pane root = createRootPane();
      Scene scene1 = new Scene(root);
      primaryStage.setScene(scene1);
      primaryStage.setTitle("Chart");
      primaryStage.show();
   }

   public Pane createRootPane()
   {
      GridPane pane = new GridPane();
      NumberAxis xAxis = new NumberAxis();
      xAxis.setLabel("Period");
      NumberAxis yAxis = new NumberAxis();
      yAxis.setLabel("Balance");

      LineChart<Number, Number> chart = new LineChart<>(xAxis, yAxis);
      XYChart.Series<Number, Number> balances1 = new XYChart.Series<>();
      balances1.setName("5%");
      final double INITIAL_BALANCE = 10000;
      final double PERIODS = 60;
      double rate = 5;
      double balance = INITIAL_BALANCE;
      for (int i = 0; i <= PERIODS; i++)
      {
         balances1.getData().add(new XYChart.Data<>(i, balance));
         balance = balance + balance * rate / 100;
      }
      chart.getData().add(balances1);

      XYChart.Series<Number, Number> balances2 = new XYChart.Series<>();
      balances2.setName("6%");
      rate = 6;
      balance = INITIAL_BALANCE;
      for (int i = 0; i <= PERIODS; i++)
      {
         balances2.getData().add(new XYChart.Data<>(i, balance));
         balance = balance + balance * rate / 100;
      }
      chart.getData().add(balances2);

      pane.add(chart, 0, 0);

      return pane;
   }
}
```

WORKED EXAMPLE 11.1 Programming a Working Calculator

In this Worked Example, we implement arithmetic and scientific operations for a calculator. We use the sample program from Section 11.1 as a starting point.

Layout

Here is how our calculator will look:



The user interface is a bit different from that in Section 11.1. As you can see, the button grid has buttons + - * / for arithmetic operations. There is also a pane at the bottom that holds a checkbox, radio buttons, and a choice box for specifying a mathematical function and its behavior. The "Apply" button is clicked to apply the selected function.

We use a grid pane layout, as in Section 11.1. However, now the grid has four columns, and the display field spans all four:

```
GridPane pane = new GridPane();
pane.setHgap(10);
pane.setVgap(10);
pane.setPadding(new Insets(10));

ColumnConstraints col1 = new ColumnConstraints();
col1.setPercentWidth(25);
ColumnConstraints col2 = new ColumnConstraints();
col2.setPercentWidth(25);
ColumnConstraints col3 = new ColumnConstraints();
col3.setPercentWidth(25);
ColumnConstraints col4 = new ColumnConstraints();
col4.setPercentWidth(25);
pane.getColumnConstraints().addAll(col1, col2, col3, col4);

display = new TextField("");
pane.add(display, 0, 0, 4, 1);
```

We then loop over the button labels and add each button:

```
String[] labels = { "7", "8", "9", "/", "4", "5", "6", "*",
      "1", "2", "3", "-", "0", ".", "=", "+" };
int row = 1;
int column = 0;
for (String label : labels)
{
   Button b = new Button(label);
   b.setMaxWidth(Double.MAX_VALUE);
   pane.add(b, column, row);
   GridPane.setHalignment(b, HPos.CENTER);
   column++;
   if (column == 4) { column = 0; row++; }
   . . .
}
```

We add the remaining controls into an `HBox` that also spans four colums. We use an `HBox` rather than placing the controls directly into the grid because we don't want each of them to have the same size as the buttons above them.

```
HBox bottomBox = new HBox(
      radianCheckBox,
      new Label("Log base"),
      baseeButton,
      base10Button,
      base2Button,
      mathOpChoice,
      mathOpButton);
 pane.add(bottomBox, 0, 5, 4, 1);
```

Arithmetic

First, we need to add button actions to the calculator pane for the arithmetic operations.

It is actually a bit subtle to implement the behavior of a calculator. Imagine the user who has just entered 3 +. At this point, we can't yet perform the addition because we don't have the second operand. We need to store the value (3) and the operator (+) and keep on going. Now the user continues:

```
3 + 4 *
```

As soon as the * button is clicked, we can get to work and *add* 3 and 4. That is, we take the saved value and the newly entered value, and combine them with the *saved* operator. Then we save the * so that it can be executed later.

(Here, we implement a common household calculator in which multiplication and addition have the same precedence. With additional effort, it is possible to implement a calculator in which multiplication has a higher precedence, as it does in mathematics.)

There is another subtlety, concerning the update of the calculator display. Consider the input

```
1 3 + 4 * 2 =
```

which arrives one button click at a time:

Button Clicked	Action	Display
1	Show 1 in display.	1
3	Add 3 to end of display.	13
+	Store 13 and + for later use.	13
4	Clear display, add 4.	4
*	Replace display with result of 13 + 4. Store 17 and * for later use.	17
2	Clear display, add 2.	2
=	Replace display with result of 17 * 2.	34

You may want to try this out with an actual calculator. Note the following:

- When an operator button is clicked and two operands are available, the display is updated with the result of the saved operation.
- The first digit button clicked after an operator clears the display. The other digit buttons append to the display. The display can't be cleared by the operator; it must be cleared by the first digit. (Otherwise, there would be no way for the user to see the result.)
- The = button puts the calculator into the same state as it was at the beginning, clearing the saved operation.

Now we have enough information to implement the arithmetic operator buttons. The calculator needs to remember

- the last value and operator.
- whether we are at the beginning or in the middle of entering a value.

We also need to remember the value that is currently being built up, but we can just take that from the display field.

```
public class Calculator extends Application
{
   private double lastValue;
   private String lastOperator;
   private boolean startNewValue;
   private TextField display;
   . . .
   public void start(Stage stage1)
   {
      lastValue = 0;
      lastOperator = "=";
      startNewValue = true;
      . . .
   }
   . . .
}
```

The event handler of the digit buttons calls the following method which appends the digit to the display. However, the display is cleared first if this was the first digit after an operator:

```
public void addToValue(String ch)
{
   if (startNewValue)
   {
      display.setText(ch);
      startNewValue = false;
   }
   else
   {
      display.setText(display.getText() + ch);
   }
}
```

The handler is set in the loop that constructs the buttons:

```
for (String label : labels)
{
   Button b = new Button(label);
   . . .
   if ("+-*/=".contains(label))
   {
      . . .
   }
   else
   {
      b.setOnAction(event -> addToValue(label));
   }
}
```

Handling the operator buttons is a bit more complex. We need to use the last operator for combining the values, remember the current operator, and remember that the next digit button should start a new value:

```
if ("+-*/=".contains(label))
{
   b.setOnAction(event ->
   {
      calculate(lastOperator);
      lastOperator = label;
      startNewValue = true;
   });
}
```

Here is the `calculate` method that computes the result and updates the display:

```
public void calculate(String op)
{
   if (!startNewValue)
   {
      double value = Double.parseDouble(display.getText());
      double result = value;
```

```
         if (op.equals("+"))
         {
            result = lastValue + value;
         }
         else if (op.equals("-"))
         {
            result = lastValue - value;
         }
         else if (op.equals("*"))
         {
            result = lastValue * value;
         }
         else if (op.equals("/"))
         {
            result = lastValue / value;
         }
         setValue(result);
      }
   }
```

In this method, we first check whether the operator follows a value. If a user clicked two operators in a row, as in `3 + * 4`, we assume that the intent was to replace an incorrectly entered operator.

Here is the `setValue` method for updating the display:

```
public void setValue(double value)
{
   display.setText("" + value);
   lastValue = value;
   startNewValue = true;
}
```

To understand the behavior for the `=` operator, think through an input `3 + 4 =` followed by `5 * 6 =`. When the `=` button is clicked for the first time, the last operator (`+`) is executed, and `=` becomes the last operator. When the `*` button is clicked, the `calculate` method should simply return the second operand (5), which will later be combined with the 6.

This completes the implementation of the arithmetic operators.

Mathematical Functions

In order to practice working with user-interface controls, the calculator supports a few mathematical functions. The trigonometric functions `sin`, `cos`, and `tan` take an argument that can be interpreted as radians or degrees. We provide a check box to select radians. (Perhaps two radio buttons for radians and degrees would be clearer, but we want to practice using a checkbox.) For the `log` and `exp` functions, we provide radio buttons to select one of three bases: e, 10, and 2. We place the functions themselves into a choice box.

Clicking the “Apply” button applies the selected function with the selected options. Here is the button handler:

```
mathOpButton.setOnAction(event ->
   {
      boolean radian = radianCheckBox.isSelected();
      double base = Math.E;
      if (base10Button.isSelected()) { base = 10; }
      else if (base2Button.isSelected()) { base = 2; }
      String functionName = mathOpChoice.getSelectionModel().getSelectedItem();
      computeMathFunction(functionName, radian, base);
   });
```

Here is the method for computing the result of the function. If we need to call a trigonometric function with degrees, we convert the argument to radians. That is what the Java library expects.

```
public void computeMathFunction(String functionName, boolean radian, double base)
{
   double value = Double.parseDouble(display.getText());
   if (!radian && (functionName.equals("sin")
      || functionName.equals("cos") || functionName.equals("tan")))
   {
      value = Math.toRadians(value);
   }

   double result = value;
   if (functionName.equals("sin"))
   {
      result = Math.sin(value);
   }
   else if (functionName.equals("cos"))
   {
      result = Math.cos(value);
   }
   else if (functionName.equals("tan"))
   {
      result = Math.tan(value);
   }
   else if (functionName.equals("log"))
   {
      result = Math.log(value) / Math.log(base);
   }
   else if (functionName.equals("exp"))
   {
      result = Math.pow(base, value);
   }
   setValue(result);
}
```

Here is the complete source code. Note that the `display` text field is an instance variable—we need to query and update its contents in several methods. However, the check box, radio buttons, and choice box are local variables that are read only in the handler for the "Apply" button.

worked_example_1/Calculator.java

```
import javafx.application.Application;
import javafx.geometry.HPos;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.ColumnConstraints;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Pane;
import javafx.stage.Stage;

public class Calculator extends Application
{
   private double lastValue;
   private String lastOperator;
   private boolean startNewValue;
   private TextField display;

   public void start(Stage stage1)
   {
      lastValue = 0;
      lastOperator = "=";
      startNewValue = true;

      Pane root = createRootPane();
      Scene scene1 = new Scene(root);
      stage1.setScene(scene1);
      stage1.setTitle(" ");
      stage1.show();
   }

   public Pane createRootPane()
   {
      GridPane pane = new GridPane();
      pane.setHgap(10);
      pane.setVgap(10);
      pane.setPadding(new Insets(10));

      ColumnConstraints col1 = new ColumnConstraints();
      col1.setPercentWidth(25);
      ColumnConstraints col2 = new ColumnConstraints();
      col2.setPercentWidth(25);
      ColumnConstraints col3 = new ColumnConstraints();
      col3.setPercentWidth(25);
      ColumnConstraints col4 = new ColumnConstraints();
      col4.setPercentWidth(25);
      pane.getColumnConstraints().addAll(col1, col2, col3, col4);

      display = new TextField("");
      pane.add(display, 0, 0, 4, 1);

      String[] labels = { "7", "8", "9", "/", "4", "5", "6", "*",
            "1", "2", "3", "-", "0", ".", "=", "+" };
      int row = 1;
      int column = 0;
      for (String label : labels)
      {
         Button b = new Button(label);
         b.setMaxWidth(Double.MAX_VALUE);
         pane.add(b, column, row);
         GridPane.setHalignment(b, HPos.CENTER);
         column++;
         if (column == 4) { column = 0; row++; }

         if ("+-*/=".contains(label))
         {
            b.setOnAction(event ->
            {
               calculate(lastOperator);
               lastOperator = label;
               startNewValue = true;
            });
         }
         else
         {
            b.setOnAction(event -> addToValue(label));
         }
      }

      CheckBox radianCheckBox = new CheckBox("Radian");

      RadioButton baseeButton = new RadioButton("e");
      RadioButton base10Button = new RadioButton("10");
      RadioButton base2Button = new RadioButton("2");

      ToggleGroup baseButtonGroup = new ToggleGroup();
      baseeButton.setToggleGroup(baseButtonGroup);
      base10Button.setToggleGroup(baseButtonGroup);
      base2Button.setToggleGroup(baseButtonGroup);
      baseeButton.setSelected(true);

      ChoiceBox<String> mathOpChoice = new ChoiceBox<>();
      mathOpChoice.getItems().addAll("sin", "cos", "tan", "log", "exp");
      mathOpChoice.getSelectionModel().select(0);
      Button mathOpButton = new Button("Apply");

      HBox bottomBox = new HBox(
         radianCheckBox,
         new Label("Log base"),
         baseeButton,
         base10Button,
         base2Button,
         mathOpChoice,
         mathOpButton);
      pane.add(bottomBox, 0, 5, 4, 1);

      mathOpButton.setOnAction(event ->
         {
            boolean radian = radianCheckBox.isSelected();
            double base = Math.E;
            if (base10Button.isSelected()) { base = 10; }
            else if (base2Button.isSelected()) { base = 2; }
            String functionName =
                  mathOpChoice.getSelectionModel().getSelectedItem();
            computeMathFunction(functionName, radian, base);
         });

      return pane;
   }

   /**
      Sets the display to a new value.
      @param value the new value
   */
   public void setValue(double value)
   {
      display.setText("" + value);
      lastValue = value;
      startNewValue = true;
   }

   /**
      Adds a character to the display value.
      @param ch the character to add
   */
```

```java
   public void addToValue(String ch)
   {
      if (startNewValue)
      {
         display.setText(ch);
         startNewValue = false;
      }
      else
      {
         display.setText(display.getText() + ch);
      }
   }

   /**
      Calculates an arithmetic operation.
      @param op the arithmetic operation for the next step
   */
   public void calculate(String op)
   {
      if (!startNewValue)
      {
         double value = Double.parseDouble(display.getText());
         double result = value;
         if (op.equals("+"))
         {
            result = lastValue + value;
         }
         else if (op.equals("-"))
         {
            result = lastValue - value;
         }
         else if (op.equals("*"))
         {
            result = lastValue * value;
         }
         else if (op.equals("/"))
         {
            result = lastValue / value;
         }
         setValue(result);
      }
   }

   /**
      Calculates a mathematical function.
      @param functionName the name of the function
      @param radian true if trigonometric functions use radian
      @param base the base for log and exp
   */
   public void computeMathFunction(String functionName,
         boolean radian, double base)
   {
      double value = Double.parseDouble(display.getText());

      if (!radian && (functionName.equals("sin")
            || functionName.equals("cos") || functionName.equals("tan")))
      {
         value = Math.toRadians(value);
      }

      double result = value;
      if (functionName.equals("sin"))
      {
         result = Math.sin(value);
      }
      else if (functionName.equals("cos"))
      {
         result = Math.cos(value);
      }
      else if (functionName.equals("tan"))
      {
         result = Math.tan(value);
      }
      else if (functionName.equals("log"))
      {
         result = Math.log(value) / Math.log(base);
      }
      else if (functionName.equals("exp"))
      {
         result = Math.pow(base, value);
      }
      setValue(result);
   }
}
```

11.3 Menus

The menu bar contains menus. A menu contains submenus and menu items.

Anyone who has ever used a graphical user interface is familiar with pull-down menus (see Figure 5). A *menu bar* contains the top-level menus. Each menu is a collection of *menu items* and *submenus*.

The sample program for this section builds up a small but typical menu and handles the events from the menu items. The program allows the user to specify the font for a label by selecting a face name, font size, and font style. In JavaFX, it is easy to create these menus.

A menu provides a list of available choices.

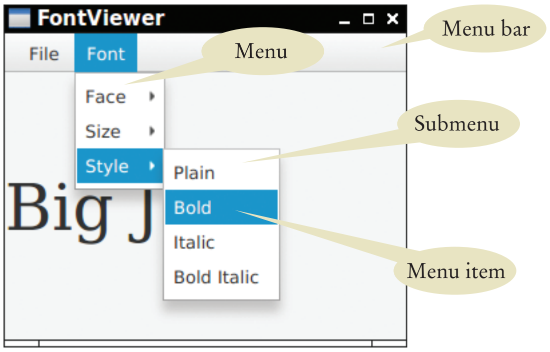


Figure 5 Pull-Down Menus

You add the menu bar to the top of the root pane:

```
private Pane createRootPane()
{
   MenuBar bar = new MenuBar();
   . . .
   VBox pane = new VBox(bar, . . .);
   return pane;
}
```

A menu provides a list of available choices.

Add menus to the menu bar:

```
Menu fileMenu = new Menu("File");
Menu fontMenu = new Menu("Font");
bar.getMenus().addAll(fileMenu, fontMenu);
```

Add menu items and submenus to the menus:

```
MenuItem exitItem = new MenuItem("Exit");
fileMenu.getItems().add(exitItem);
Menu styleMenu = new Menu("Style");
fontMenu.getItems().add(styleMenu); // A submenu
```

Menu items generate action events.

A menu item has no further submenus. When the user selects a menu item, the menu item sends an action event. Therefore, you want to add a handler to each menu item:

```
exitItem.setOnAction(event -> System.exit(0));
```

You add action event handlers only to menu items, not to menus or the menu bar. When the user clicks on a menu name and a submenu opens, no action event is sent.

To keep the program readable, it is a good idea to use methods for constructing menu items with similar actions. For example,

```
private MenuItem createFaceItem(String newFacename)
{
   MenuItem item = new MenuItem(newFacename);
   item.setOnAction(event -> { facename = newFacename; updateSample(); });
   return item;
}
```

Here, `facename` is an instance variable of the `FontViewer` class, and the `updateSample` method uses the face name and other font information to update the font sample:

```
public class FontViewer extends Application
{
   private Label sample;
   private String facename = "Serif";
   . . .
   private void updateSample()
   {
      sample.setFont(Font.font(facename, . . .));
   }
}
```

Here is the complete program:

sec03/FontViewer.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.Menu;
import javafx.scene.control.MenuBar;
import javafx.scene.control.MenuItem;
import javafx.scene.layout.Pane;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.scene.text.FontPosture;
import javafx.scene.text.FontWeight;
import javafx.stage.Stage;

public class FontViewer extends Application
{
   private static final int SMALL_SIZE = 24;
   private static final int MEDIUM_SIZE = 36;
   private static final int LARGE_SIZE = 48;

   private Label sample;
   private String facename = "Serif";
   private int size = LARGE_SIZE;
   private FontPosture posture = FontPosture.REGULAR;
   private FontWeight weight = FontWeight.NORMAL;

   public void start(Stage primaryStage)
   {
      Pane root = createRootPane();
      Scene scene1 = new Scene(root);
      primaryStage.setScene(scene1);
      primaryStage.setTitle("FontViewer");
      primaryStage.show();
   }

   private MenuItem createFaceItem(String newFacename)
   {
      MenuItem item = new MenuItem(newFacename);
      item.setOnAction(event -> { facename = newFacename; updateSample(); });
      return item;
   }

   private MenuItem createSizeItem(String name, int newSize)
   {
      MenuItem item = new MenuItem(name);
      item.setOnAction(event -> { size = newSize; updateSample(); });
      return item;
   }

   private MenuItem createStyleItem(String name,
      FontPosture newPosture, FontWeight newWeight)
   {
      MenuItem item = new MenuItem(name);
      item.setOnAction(event ->
         {
            posture = newPosture;
            weight = newWeight;
            updateSample();
         });
      return item;
   }

   private Pane createRootPane()
   {
      MenuBar bar = new MenuBar();
      sample = new Label("Big Java");
      sample.setMinSize(300, 200);
```

```java
        VBox pane = new VBox(bar, sample);
        Menu fileMenu = new Menu("File");
        MenuItem exitItem = new MenuItem("Exit");
        exitItem.setOnAction(event -> System.exit(0));
        fileMenu.getItems().add(exitItem);

        Menu fontMenu = new Menu("Font");
        bar.getMenus().addAll(fileMenu, fontMenu);

        Menu faceMenu = new Menu("Face");
        faceMenu.getItems().addAll(
              createFaceItem("Serif"),
              createFaceItem("SansSerif"),
              createFaceItem("Monospaced"));

        Menu sizeMenu = new Menu("Size");
        sizeMenu.getItems().addAll(
              createSizeItem("Small", SMALL_SIZE),
              createSizeItem("Medium", MEDIUM_SIZE),
              createSizeItem("Large", LARGE_SIZE));

        Menu styleMenu = new Menu("Style");
        styleMenu.getItems().addAll(
              createStyleItem("Plain", FontPosture.REGULAR, FontWeight.NORMAL),
              createStyleItem("Bold", FontPosture.REGULAR, FontWeight.BOLD),
              createStyleItem("Italic", FontPosture.ITALIC, FontWeight.NORMAL),
              createStyleItem("Bold Italic", FontPosture.ITALIC,
                    FontWeight.BOLD));

        fontMenu.getItems().addAll(faceMenu, sizeMenu, styleMenu);

        updateSample();
        return pane;
    }

    private void updateSample()
    {
        sample.setFont(Font.font(facename, weight, posture, size));
    }
}
```

SELF CHECK

11. Why do `Menu` objects not generate action events?
12. Can you add a menu item directly to the menu bar? Try it out. What happens?
13. Can you add a menu to itself as child menu?
14. Why can't the `createFaceItem` method simply set the `faceName` instance variable, like this:

```java
private MenuItem createFaceItem(String newFacename)
{
   MenuItem item = new MenuItem(newFacename);
   facename = newFacename;
   item.setOnAction(event -> { updateSample(); });
   return item;
}
```

15. In this program, the font specification (name, size, and style) is stored in instance variables. Why was this not necessary in the program of the previous section?

Practice It Now you can try these exercises at the end of the chapter: R11.13, E11.7, E11.8.

11.4 Properties and Bindings

A property is accessed by getter and setter methods.

A *property* is a named attribute of a class that you can read or write. For example, the `Label` class has a property named `text` that you can read and write with the methods

```
String getText()
void setText(String value)
```

Does that mean that the `Label` has an instance variable `text`? Not necessarily. The `Label` is free to store that information any way it chooses, perhaps inside another object. For a property, all that matters are the getter and setter methods.

Each property has a name and a type.

Each property has a type, namely the return type of the getter and the parameter type of the setter. For example, the `text` property of the `Label` class has type `String` because the `getText` method returns `String` values, and the `setText` method accepts `String` values. If the property has type `boolean`, the getter method starts with `is`. For example, you call the `isSelected` getter method for the `selected` property of a `Checkbox`.

An observable property notifies its listeners when its value changes.

A property is *observable* if you can add a handler that is notified whenever the property value changes. Many properties of JavaFX controls are observable, including the `text` property of the `Label` class. Admittedly, it is not so interesting to install a handler to find out when the label text changes. You know when that happens: when you call the `setText` method. However, consider a `Slider` (see Figure 6). It has a `value` property that changes when the user adjusts the slider position.

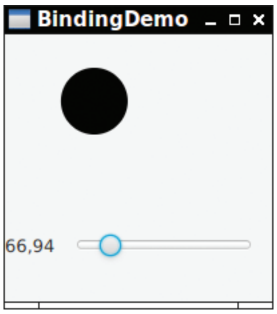


Figure 6 When the Slider is Moved, the Label and Circle are Updated

To be notified of the changes, get the property object and attach an event handler:

```
Slider positionSlider = new Slider(50, 150, 100);
   // Values range from 50 to 150, starting with 100
positionSlider.valueProperty().addListener(
      obs -> positionLabel.setText("" + positionSlider.getValue()));
```

Now the label changes whenever the slider is moved.

The event handler receives the observable that has changed, but you don't normally need it. Just ignore it and get the changed value with the property getter.

Attach property change handlers to `Property` objects.

You always attach the handler to a property object, which is returned by calling a method whose name starts with the property name and is followed by `Property`. For example, the `valueProperty` method yields an object representing the `value` property.

When the types of two properties match, you can *bind* them together. Then a change in one property automatically changes the other. Consider this example:

```
Circle ball = new Circle(100, 50, 25);
ball.centerXProperty().bind(positionSlider.valueProperty());
```

When a property is bound to another, it tracks the changes of the other property.

When the user moves the slider, the circle moves with it. You could have achieved the same effect with a listener:

```
positionSlider.valueProperty().addListener(
      obs -> ball.setCenterX(positionSlider.getValue()));
```

But the `bind` syntax is more compact.

Note that a property can only be bound to one other property. There is also an `unbind` method to remove the binding.

You can only bind properties together that have the same type. For example, we cannot directly bind the slider's `value` property (of type `double`) to the `text` property of a label (which has type `String`). You can convert a numeric property to a string property with the `asString` method. Call the method with a format specifier of the style used by `String.format`:

```
positionLabel.textProperty().bind(positionSlider.valueProperty().asString("%.2f"));
```

sec04/BindingDemo.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.Slider;
import javafx.scene.layout.Pane;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class BindingDemo extends Application
{
   public void start(Stage primaryStage)
   {
      Pane root = createRootPane();
      Scene scene1 = new Scene(root);
      primaryStage.setScene(scene1);
      primaryStage.setTitle("BindingDemo");
      primaryStage.show();
   }

   public Pane createRootPane()
   {
      Circle ball = new Circle(100, 50, 25);
      Slider positionSlider = new Slider(50, 150, 100);
      Label positionLabel = new Label("");

      Pane pane = new Pane(ball, positionSlider, positionLabel);
      pane.setMinSize(200, 200);
      positionSlider.relocate(50, 150);
      positionLabel.relocate(0, 150);

      ball.centerXProperty().bind(positionSlider.valueProperty());
      positionSlider.valueProperty().addListener(
            obs -> positionLabel.setText("" + positionSlider.getValue()));
      // This also works:
      // positionLabel.textProperty().bind(
      //    positionSlider.valueProperty().asString("%.2f"));

      return pane;
   }
}
```

If you want to bind a property to the result of a complex computation, use the static `Bindings.createObjectBinding` method. You supply a lambda expression for computing the desired result, followed by one or more properties. If any of the properties change their value, the lambda expression is executed, and the target property is set to the result. For example, suppose you have three sliders, one each for the red, green, and blue values of a color. Then you can bind the background color property of a circle like this:

```
ball.fillProperty().bind( // This property is set ...
   Bindings.createObjectBinding(
      () ->
         Color.rgb((int) redSlider.getValue(), // ... to the result of this expression ...
         (int) greenSlider.getValue(),
         (int) blueSlider.getValue()),
      redSlider.valueProperty(), // ... when one of these properties changes.
      greenSlider.valueProperty(),
      blueSlider.valueProperty()));
```

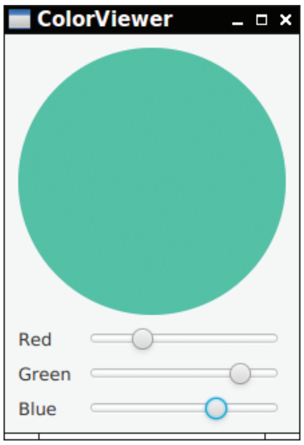


Figure 7 A Color Viewer with Sliders

Here is the complete color viewer program. For the layout, we use a grid layout with two columns. In this example, we do not want the columns to have equal size. The circle spans both columns. Its color changes as the sliders are adjusted.

sec04/ColorViewer.java

```
import javafx.application.Application;
import javafx.beans.binding.Bindings;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
```

```java
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class ColorViewer extends Application
{
   public void start(Stage primaryStage)
   {
      Pane root = createRootPane();
      Scene scene1 = new Scene(root);
      primaryStage.setScene(scene1);
      primaryStage.setTitle("ColorViewer");
      primaryStage.show();
   }

   public Pane createRootPane()
   {
      Circle ball = new Circle(150, 150, 100);
      GridPane pane = new GridPane();
      pane.setHgap(10);
      pane.setVgap(10);
      pane.setPadding(new Insets(10));

      pane.add(ball, 0, 0, 2, 1);
      pane.add(new Label("Red"), 0, 1);
      pane.add(new Label("Green"), 0, 2);
      pane.add(new Label("Blue"), 0, 3);
      Slider redSlider = new Slider(0, 255, 0);
      Slider greenSlider = new Slider(0, 255, 0);
      Slider blueSlider = new Slider(0, 255, 0);
      pane.add(redSlider, 1, 1);
      pane.add(greenSlider, 1, 2);
      pane.add(blueSlider, 1, 3);

      ball.fillProperty().bind(
            Bindings.createObjectBinding( // A computed property
                  () -> Color.rgb((int) redSlider.getValue(), // Call this ...
                        (int) greenSlider.getValue(),
                        (int) blueSlider.getValue()),
                  redSlider.valueProperty(), // ... when one of these changes
                  greenSlider.valueProperty(),
                  blueSlider.valueProperty()));

      return pane;
   }
}
```

SELF CHECK

16. A `Font` object has a size. Is that a property?
17. A `Label` has a font. Is that a property? Is it observable?
18. Why doesn't a slider emit action events?
19. What do you need to change in the `BindingDemo` program to make the slider change the size of the circle?
20. Suppose we want to replace sliders with text fields in the `ColorViewer` program, so that a user can enter the red, green, and blue values, and the background color is updated as the text changes. How do you set up the binding?

Practice It Now you can try these exercises at the end of the chapter: R11.14, E11.10, E11.11.

Special Topic 11.2

Custom Properties

JavaFX controls have many properties, and you can use them without knowing how they are implemented. However, for advanced user interfaces, as well as for animations (which we will discuss in the following section), you may want to expose properties of your own classes.

Consider the `Bar` class in Worked Example 10.1. You may want to adjust the size of a bar by binding it to a slider. Then you need to supply a `size` property. That property has type `double`.

Therefore, you need to provide methods

```
public double getSize()
public void setSize(double value)
```

as well as a method `sizeProperty` that returns a property object.

You need to choose a class for the property. The most commonly used ones are:

- `IntegerProperty`, `DoubleProperty`, `BooleanProperty` for properties whose type is a primitive type.
- `StringProperty` for properties whose type is `String`.
- `ObjectProperty<T>` for properties of any other type `T`.

In our case, the `sizeProperty` method should return a `DoubleProperty` object:

```
public DoubleProperty sizeProperty()
```

The JavaFX API provides classes `SimpleIntegerProperty`, `SimpleDoubleProperty`, and so on, that provide a simple mechanism for implementing any property. Here is how to do that:

```
public class Bar
{
   private Rectangle rect1;
   private DoubleProperty size;

   public Bar(String label, double initialSize)
   {
      size = new SimpleDoubleProperty(initialSize);
      . . .
   }
   . . .
   public double getSize() { return size.get(); }
   public void setSize(double value) { size.set(value); }
   public DoubleProperty sizeProperty() { return size; }
   . . .
}
```

The `DoubleProperty` object contains a `double` value, which you can get and set. It also provides the mechanism for adding listeners.

Now you can bind the property:

```
Bar bar1 = new Bar(. . .);
bar1.sizeProperty().bind(positionSlider.valueProperty());
```

There is just one problem. The rectangle that makes up the bar isn't changing yet. You need to add a listener to the property object that updates the rectangle width whenever the property value changes. As it happens, the rectangle width is a property of the same type, so you can just bind it:

```
public Bar(String label, double initialSize)
{
   . . .
   rect1.widthProperty().bind(size);
}
```

If there was a more complicated relationship, you would install a listener or create an object binding.

Also, any other methods that affect the rectangle width must update the property, not the rectangle. For example, suppose we have a growSize method:

```
public void growSize(double dx)
{
   rect1.setWidth(rect1.getWidth() + dx);
}
```

Change it to:

```
public void growSize(double dx)
{
   setSize(getSize() + dx);
}
```

special_topic_2/CustomPropertyDemo.java

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Slider;
import javafx.scene.layout.Pane;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class CustomPropertyDemo extends Application
{
   public void start(Stage primaryStage)
   {
      Pane root = createRootPane();
      Scene scene1 = new Scene(root);
      primaryStage.setScene(scene1);
      primaryStage.setTitle("CustomPropertyDemo");
      primaryStage.show();
   }

   public Pane createRootPane()
   {
      Slider slider1 = new Slider(0, 200, 100);
      Bar bar1 = new Bar("January Sales", 100);

      VBox pane = new VBox(10, slider1, bar1);
      pane.setPadding(new Insets(10));
      pane.setMinWidth(300);

      bar1.sizeProperty().bind(slider1.valueProperty());

      return pane;
   }
}
```

special_topic_2/Bar.java

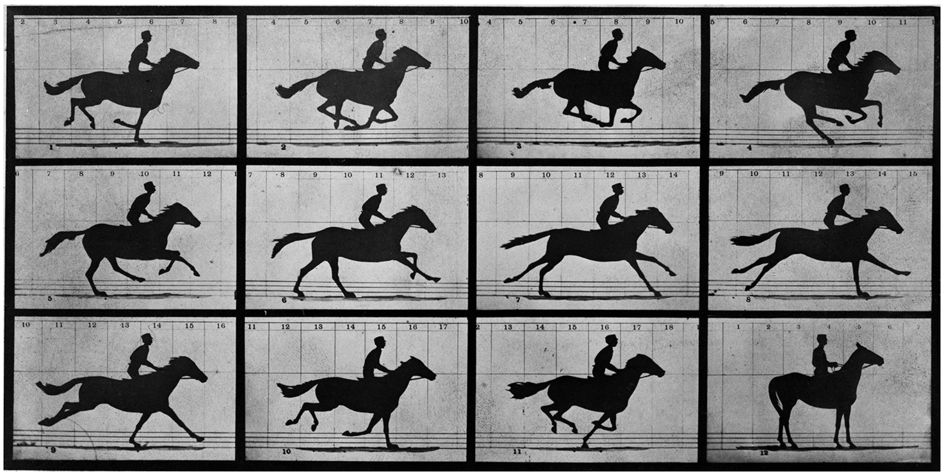
```
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Text;

public class Bar extends Pane
{
   public static final int HEIGHT = 15;
   private Rectangle rect1;

   private DoubleProperty size;
   public DoubleProperty sizeProperty() { return size; }
   public double getSize() { return size.get(); }
   public void setSize(double value) { size.set(value); }

   public Bar(String label, double initialSize)
   {
      rect1 = new Rectangle(0, 0, initialSize, HEIGHT);
      size = new SimpleDoubleProperty(initialSize);
      rect1.widthProperty().bind(size);

      Text text1 = new Text(label);
      text1.relocate(0, 0);
      text1.setStroke(Color.WHITE);

      getChildren().addAll(rect1, text1);
   }

   public void growSize(double dx)
   {
      setSize(getSize() + dx);
   }
}
```

11.5 Animations

The "FX" in JavaFX stands for "effects", and one design goal for the JavaFX library was to simplify programming of special effects such as animations.

Key frames specify actions whose intermediate steps can be interpolated.

When planning an animation, the designer produces a sequence of "key frames" that specify when actions begin and end. The movie frames that come in between can be interpolated from the two key frames that surround it.

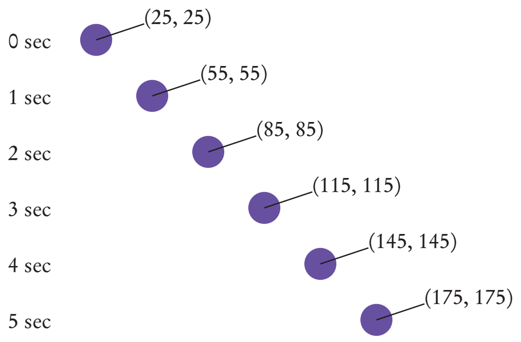


E. Muybridge's *Sallie Gardner at a Gallop*

JavaFX uses a similar terminology. A `KeyFrame` has a given duration, plus one or more properties with their target values. Here is an example:

```
KeyFrame frame1 = new KeyFrame(Duration.seconds(5),
   new KeyValue(ball.centerXProperty(), 175),
   new KeyValue(ball.centerYProperty(), 175));
```

Over five seconds, both the `centerX` and `centerY` property of `ball` change from their initial value to 175. That will make the circle move smoothly from its initial position toward the bottom-right of the pane, as shown below.



You add `KeyFrame` objects to a `Timeline`, and then you invoke the `play` method:

```
Timeline animation = new Timeline(frame1, frame2, frame3);
animation.play();
```

A timeline plays key frames in parallel.

All key frames are played in parallel. Each frame may have a different duration. The animation ends when the one with the longest duration has finished.

Use a `SequentialTransition` to play one timeline after another.

To play multiple timelines in sequence, join them to a `SequentialTransition`, and play it:

```
SequentialTransition animation =
   new SequentialTransition(timeline1, timeline2, timeline3);
animation.play();
```

By default, the `play` method plays an animation once. To repeat the animation, call:

```
animation.setCycleCount(n);
```

If you set `n` to `Animation.INDEFINITE`, then the animation repeats until it is stopped by a call to the `stop` method.

You can set the `autoReverse` property of an animation to run it backwards in every second cycle. For example, if an `animation` moves a circle from one place to another, then the following statements make the circle move back and forth five times:

```
animation.setCycleCount(10);
animation.setAutoReverse(true);
animation.play();
```

The key frames that you have seen here work well for a *linear* interpolation of properties, such as moving an object from one position to another at a constant speed. However, they are not as well suited for modeling nonlinear phenomena, such as accelerating objects, pendulums, or springs.

In those cases, you need to provide a mathematical formula that tells how to update objects, and you apply it many times per second.

You achieve that with a key frame that has a very short duration and no properties to update. Add an action event handler that is called when the key frame has completed. Put the key frame into a timeline that is repeated many times:

```
long start = System.currentTimeMillis();
KeyFrame frame1 = new KeyFrame(Duration.millis(10),
   e ->
      {
         double t = System.currentTimeMillis() - start;
         Update your objects.
      });
Timeline animation = new Timeline(frame1);
animation.setCycleCount(n);
animation.play();
```

In the action event handler, use the elapsed time `t` to compute the new position of the objects in motion. You can see an example in the following program. Clicking the “Nonlinear” button shows an animation that simulates a falling ball, accelerated by gravity. The program also demonstrates linear interpolation, sequential transitions, and cycles.

sec05/AnimationDemo.java

```java
import javafx.animation.KeyFrame;
import javafx.animation.KeyValue;
import javafx.animation.SequentialTransition;
import javafx.animation.Timeline;
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.Pane;
import javafx.scene.layout.VBox;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
import javafx.util.Duration;

public class AnimationDemo extends Application
{
   public void start(Stage primaryStage)
   {
      Pane root = createRootPane();
      Scene scene1 = new Scene(root);
      primaryStage.setScene(scene1);
      primaryStage.setTitle("AnimationDemo");
      primaryStage.show();
   }

   public Pane createRootPane()
   {
      Circle ball = new Circle(25, 50, 25);

      Pane ballPane = new Pane(ball);
      ballPane.setMinSize(200, 200);

      Button keyFrames = new Button("Key frames");
      Button sequential = new Button("Sequential");
      Button cycling = new Button("Cycling");
      Button nonlinear = new Button("Nonlinear");

      VBox root = new VBox(10, ballPane, keyFrames, sequential,
         cycling, nonlinear);
      root.setPadding(new Insets(10));

      /*
         This animation simultaneously moves the circle diagonally
         within 5 seconds and shrinks the circle within 10 seconds.
      */
      keyFrames.setOnAction(event ->
         {
            ball.setCenterX(25);
            ball.setCenterY(50);
            ball.setRadius(25);

            KeyFrame frame1 = new KeyFrame(Duration.seconds(5),
               new KeyValue(ball.centerXProperty(), 175),
               new KeyValue(ball.centerYProperty(), 175));
            KeyFrame frame2 = new KeyFrame(Duration.seconds(10),
               new KeyValue(ball.radiusProperty(), 5));
            Timeline animation = new Timeline(frame1, frame2);
            animation.play();
         });

      /*
         This animation moves the circle diagonally within 5 seconds
         and then shrinks the circle within 10 seconds.
      */
      sequential.setOnAction(event ->
         {
            ball.setCenterX(25);
            ball.setCenterY(50);
            ball.setRadius(25);
            KeyFrame frame1 = new KeyFrame(Duration.seconds(5),
                  new KeyValue(ball.centerXProperty(), 175),
                  new KeyValue(ball.centerYProperty(), 175));
            KeyFrame frame2 = new KeyFrame(Duration.seconds(10),
                  new KeyValue(ball.radiusProperty(), 5));
            Timeline timeline1 = new Timeline(frame1);
            Timeline timeline2 = new Timeline(frame2);
            SequentialTransition animation =
                   new SequentialTransition(timeline1, timeline2);
            animation.play();
         });

      /*
         This animation moves the circle to the right, then reverses
         so that it moves back to the left, for a total of ten cycles.
      */
      cycling.setOnAction(event ->
         {
            ball.setCenterX(25);
            ball.setCenterY(50);
            ball.setRadius(25);
            KeyFrame frame1 = new KeyFrame(Duration.seconds(1),
                  new KeyValue(ball.centerXProperty(), 175));
            Timeline animation = new Timeline(frame1);
            animation.setCycleCount(10);
            animation.setAutoReverse(true);
            animation.play();
         });

      /*
         This animation simulates a ball falling under gravity.
      */
      nonlinear.setOnAction(event ->
         {
            ball.setCenterX(25);
            ball.setCenterY(50);
            ball.setRadius(25);

            long start = System.currentTimeMillis();
            KeyFrame frame1 = new KeyFrame(Duration.millis(10),
               e -> {
                  double t = System.currentTimeMillis() - start;
                  ball.setCenterY(50 + t * t / 100000);
               });
            Timeline animation = new Timeline(frame1);
            animation.setCycleCount(500);
            animation.play();
         });

      return root;
   }
}
```

SELF CHECK

21. How do you animate a circle whose radius grows from 100 to 200 in 2 seconds?

22. How do you animate a square whose side length grows from 100 to 200 in 2 seconds?

23. How do you animate a circle whose radius grows from 100 to 200 in 2 seconds and then shrinks to zero in the next four seconds?

24. How do you animate a circle that pulsates indefinitely, with radius growing from 100 to 110 and then shrinking back?

25. The motion of a heavy ball that is suspended from a bungee cord is described by the equation $y = y_0 + m \sin(f\,t)$, where y is the vertical position, y_0 is the point at rest, m is the maximum displacement, and f is a constant depending on the cord material. How can you implement an animation of the ball?

Practice It Now you can try these exercises at the end of the chapter: E11.14, E11.15, E11.16.

11.6 Mouse Events

You use a mouse event handler to capture mouse events.

If you write programs that show drawings, and you want users to manipulate the drawings with a mouse, then you need to handle mouse events. You can install handlers into any nodes, such as geometric shapes, buttons, or panes. There are handlers for different kinds of events. Let us start with the simplest kind: pressing the mouse button.

© james Brey/iStockphoto.

In JavaFX, a mouse event isn't a gathering of rodents; it's notification of a mouse click by the program user.

Suppose you want to move a circle to the position where a user pressed a mouse button. In the pane where you listen to mouse events, add a handler:

```
pane.setOnMousePressed(event ->
   {
      ball.setCenterX(event.getX());
      ball.setCenterY(event.getY());
   });
```

When the mouse is pressed, the handler is invoked. The handler receives a `MouseEvent`. You can get the mouse position by calling the `getX` and `getY` methods.

There are other events that you can capture. The `setOnMouseEntered` and `setOnMouseExited` methods install handlers that are called whenever the mouse enters or exits the node. You can use those handlers to highlight a node that rests under the mouse cursor, and to turn off highlighting when the mouse moves away.

You can install a handler with the `setOnMouseMoved` method that tracks whether the mouse has moved without any button presses. It is more common to use the `setOnMouseDragged` method to install a handler that is notified when the mouse is moved as a button is pressed. Such a handler can update the position of a node to show how it is being dragged.

In the following program, the user can click anywhere in a pane, and a circle is moved to the position of the mouse click. Alternatively, the user can drag the circle to a new location. As the circle is dragged, its color is changed to blue, as shown in Figure 8. When the mouse is released, the color reverts to black.

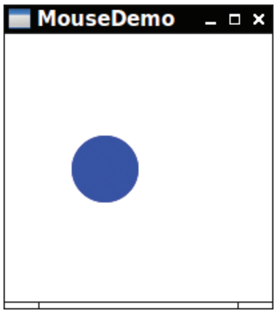


Figure 8 When the Ball is Dragged, It Turns Blue

sec06/MouseDemo.java

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class MouseDemo extends Application
{
   public void start(Stage primaryStage)
   {
      Pane root = createRootPane();
      Scene scene1 = new Scene(root);
      primaryStage.setScene(scene1);
      primaryStage.setTitle("MouseDemo");
      primaryStage.show();
   }

   public Pane createRootPane()
   {
      Circle ball = new Circle(100, 50, 25);

      Pane root = new Pane(ball);
      root.setMinSize(200, 200);
      root.setOnMousePressed(event ->
         {
            ball.setCenterX(event.getX());
            ball.setCenterY(event.getY());
         });

      root.setOnMouseDragged(event ->
         {
            ball.setFill(Color.BLUE);
            ball.setCenterX(event.getX());
            ball.setCenterY(event.getY());
         });

      root.setOnMouseReleased(event ->
         {
            ball.setFill(Color.BLACK);
         });

      return root;
   }
}
```

SELF CHECK

26. Suppose you want to add a new circle to a pane whenever the user presses the mouse button. How do you do that?

27. Suppose you want the circle in the `MouseDemo` program to light up in yellow whenever the mouse hovers over it. How do you achieve that?

28. The `MouseDemo` program has a circle that is moved to the mouse position. What changes do you have to make if you want to move a square instead?

29. Suppose you change the statement

```
root.setOnMouseDragged(. . .)
```

to

```
ball.setOnMouseDragged(. . .)
```

in the `MouseDemo` program. What is the effect? Try it out if you are not sure.

30. Suppose you change the statement

```
root.setOnMouseReleased(. . .)
```

to

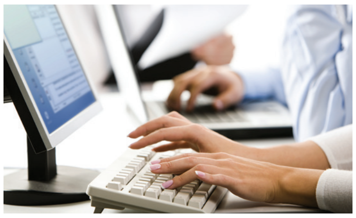
```
ball.setOnMouseReleased(. . .)
```

in the `MouseDemo` program. What is the effect? Try it out if you are not sure.

Practice It Now you can try these exercises at the end of the chapter: R11.24, E11.22, E11.26.

Special Topic 11.3

Keyboard Events



Whenever the program user presses a key, a key event is generated.

In some programs, you need to process keystrokes. For example, you may want to allow users to move a game character with the arrow keys. In the same way that you use a mouse event handler to track mouse events, you can set a key event handler to receive key event notifications. However, there is a twist. A mouse event is sent to the control under the mouse pointer. But which control should receive key events? It does not make sense to send key events to all controls—this would clearly not work if you have two text fields in your application. Instead, at most one control at a time has *focus*. A program user can change the focus by clicking with the mouse, or by using the Tab and Shift+Tab keys. The focused control is highlighted—in this case, with a blue outline; see the text field in Figure 9.

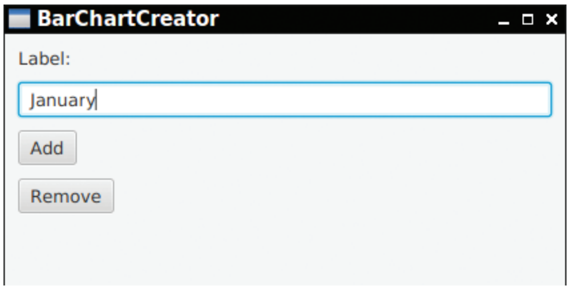


Figure 9 The Focused Control Receives Key Events

Suppose a game displays shapes in a pane. For the pane to be able to receive key events, it must declare that it can have focus, and then it should request focus:

```
chart.setFocusTraversable(true);
chart.requestFocus();
```

Next, add a key press handler to the pane. The handler receives a `KeyEvent`. Call its `getCode` method to find out which key was pressed. The `KeyCode` enumeration has constants for each key on the keyboard.

If you decide to handle a particular key event, you should *consume* it, so that the keystroke is not used for other purposes. This is particularly important for the cursor and tab keys that can be used to change the focus. Follow this outline for a key pressed handler:

```
pane.setOnKeyPressed(event ->
   {
      if (event.getKeyCode() == KeyCode.UP)
      {
         Handle the up arrow key.
         event.consume();
      }
      else if (event.getKeyCode == . . .)
      {
         . . .
      }
   });
```

The program in Worked Example 11.2 demonstrates how to handle keyboard events.

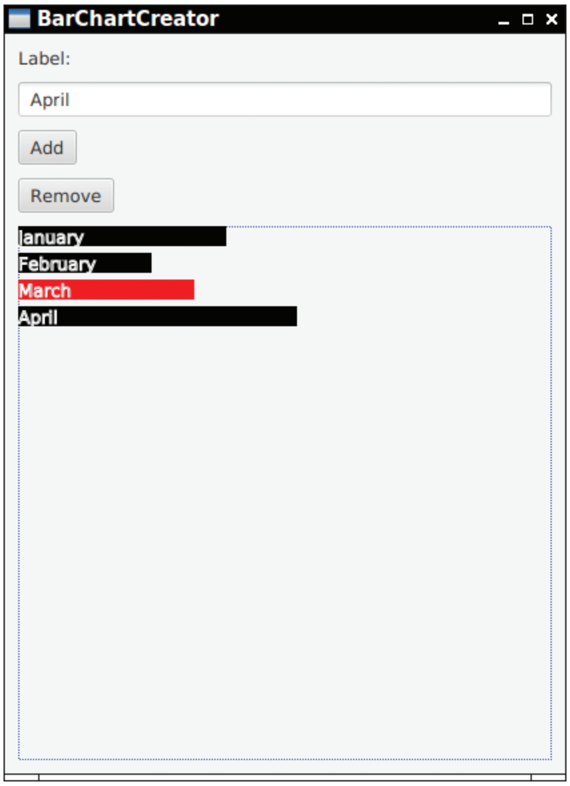
WORKED EXAMPLE 11.2

Adding Mouse and Keyboard Support to the Bar Chart Creator

In this Worked Example, we enhance the bar chart creator of Worked Example 10.1 and add support for mouse and keyboard operations.

We will implement the following user interface for modifying the chart:

- If the user selects one of the bars by clicking on it, the selected bar is displayed in red.
- If the user clicks and drags inside the row of a bar, the bar is resized so that it extends to the location of the mouse click.
- Users can also use the arrow keys. The left and right arrow keys resize the selected bar. The up and down arrow keys move to select the bar immediately above or below.



You may want to run the example program to get a feel for these operations. You have to pay close attention when you execute the keyboard commands. The chart must have focus in order to receive them. The application is made up of five controls: a label, a text field, two buttons, and the chart. When the application first starts, the text field has focus. That is, if you type keys on the keyboard, they are directed to the text field. Try it out: Type a few letters, then the left and right arrow keys. The arrow keys move the cursor. Now press the Tab key. The focus is transferred to the first button.

Press the space bar. That's the same as clicking the button. A bar is added to the chart. Now press Tab again to shift focus to the next button. Then press Tab one more time. Now the focus is on the chart, and you can use the arrow keys to adjust the bar widths.

This may seem like a tedious way of navigating a user interface. However, there are many people with a disability that prevents them from using a mouse effectively. These people rely on keyboard navigation. The JavaFX library provides the focus mechanism—you need not worry about handling Tab key presses. In this worked example, we add the keyboard shortcuts to the chart so that all users have a convenient way of editing the chart.

Updating the BarChartCreator Class

To implement these enhancements, we first modify the `BarChartCreator` and add mouse and key handlers. (See Special Topic 11.3 for handling key events.)

```
public class BarChartCreator extends Application
{
   . . .
   private Pane createRootPane()
   {
      . . .
      chart.setOnMousePressed(event ->
         {
            chart.selectBarAt(event.getY());
            chart.requestFocus();
         });

      chart.setOnMouseDragged(event ->
         {
            chart.setSelectedSize(event.getX());
         });

      chart.setOnKeyPressed(event -> {
         if (event.getCode() == KeyCode.RIGHT)
         {
            chart.growSelectedSize(1);
            event.consume();
         }
         else if (event.getCode() == KeyCode.LEFT)
         {
            chart.growSelectedSize(-1);
            event.consume();
         }
         else if (event.getCode() == KeyCode.UP)
         {
            chart.moveSelection(-1);
            event.consume();
         }
         else if (event.getCode() == KeyCode.DOWN)
         {
            chart.moveSelection(1);
            event.consume();
         }
      });
      . . .
   }
}
```

As you can see, we keep these handlers as simple as possible, and leave it to the `BarChart` class to carry out the work. On the other hand, the `BarChart` class is not concerned at all with event handling. This is a good separation of labor that you should employ in your own programs.

Now we need to implement focus handling. First, the chart should be focus traversable:

```
chart.setFocusTraversable(true);
```

Otherwise, program users won't be able to use the Tab key to give focus to the chart.

The chart should request focus when the mouse is pressed on it. We added that to the mouse event handler that you just saw.

Moreover, when a user adds a new bar, it is nice to give focus to the chart so that the user can adjust the bar width with the arrow keys. This is done in the handler for the add button:

```
addButton.setOnAction(event ->
   {
      chart.append(labelField.getText(), DEFAULT_VALUE);
      chart.requestFocus();
   });
```

Finally, we want to provide a visual indication when the chart has focus. We set a dotted border when the chart becomes focused, and turn it off when the focus is lost:

```
chart.focusedProperty().addListener(obs ->
   {
      if (chart.isFocused())
      {
         chart.setStyle("-fx-border-style: dotted;"
            + " -fx-border-width: 1px;"
            + " -fx-border-color: blue;");
      }
      else
      {
         chart.setStyle("-fx-border-style: none");
      }
   });
```

See Special Topic 11.1 for an explanation of CSS styles.

Updating the Bar and BarChart Classes

In Worked Example 10.1, a bar never changed. Now bars can be selected and unselected, and the width can be adjusted.

A selected bar is drawn in red. We provide a method that updates the bar color:

```
public void setSelected(boolean selected)
{
   if (selected)
   {
      rect1.setFill(Color.RED);
   }
   else
   {
      rect1.setFill(Color.BLACK);
   }
}
```

The bar width can be adjusted. We provide a setter as well as a method to grow or shrink the bar by a given amount. The latter method is called when the arrow keys are used to adjust the bar width.

```
public void setSize(double x)
{
   rect1.setWidth(x);
}

public void growSize(double dx)
{
   rect1.setWidth(rect1.getWidth() + dx);
}
```

The BarChart class must remember the index of the selected bar. We provide helper methods to yield the selected bar, and to move the selected bar to a new index. Note that no bar is selected when the index is –1.

```
public class BarChart extends Pane
{
   private int selectedIndex;
   . . .
   public BarChart()
   {
      selectedIndex = -1;
      . . .
   }

   private Bar getSelectedBar()
   {
      if (selectedIndex >= 0)
      {
         return (Bar) getChildren().get(selectedIndex);
      }
      else
      {
         return null;
      }
   }

   private void setSelectedIndex(int value)
   {
      int bars = getChildren().size();
      if (0 <= selectedIndex && selectedIndex < bars)
      {
         getSelectedBar().setSelected(false);
      }

      selectedIndex = value;
      if (selectedIndex < -1) { selectedIndex = -1; }
      else if (selectedIndex >= bars) { selectedIndex = bars - 1; }

      if (selectedIndex != -1)
      {
         getSelectedBar().setSelected(true);
      }
   }
   . . .
}
```

When a user clicks on the chart with the mouse, then we want to select the bar that falls on the *y*-position of the mouse. The BarChartCreator has no knowledge of the bar positions. It simply passes the *y*-position to the selectBarAt method, which determines the correct bar index:

```
public void selectBarAt(double y)
{
   setSelectedIndex((int) (y / (Bar.HEIGHT + GAP)));
}
```

The moveSelection method is used by the keyboard interface to change the selected index to a bar above or below, as indicated by the number of times the up or down arrow is pressed:

```
public void moveSelection(int by)
{
   setSelectedIndex(selectedIndex + by);
}
```

The setSelectedSize and growSelectedSize methods adjust the size of the selected bar. These methods are called when using the mouse or arrow keys to update the bar size. The methods call the corresponding methods on the selected Bar object.

```
public void setSelectedSize(double x)
{
   if (selectedIndex >= 0)
   {
      getSelectedBar().setSize(x);
   }
}

public void growSelectedSize(int dx)
{
   if (selectedIndex >= 0)
   {
      getSelectedBar().growSize(dx);
   }
}
```

The append method changes slightly to select the newly added bar:

```
public void append(String label, double value)
{
   . . .
   setSelectedIndex(bars);
}
```

In Worked Example 10.1, the removeLast method removed the last bar. Now we remove the selected bar, and we need to shift the remaining bars up.

```
public void remove()
{
   if (selectedIndex >= 0)
   {
     getChildren().remove(selectedIndex);

      int bars = getChildren().size();
      for (int i = selectedIndex; i < bars; i++)
      {
         getChildren().get(i).relocate(0, (Bar.HEIGHT + GAP) * i);
      }
      setSelectedIndex(selectedIndex);
   }
}
```

That completes the implementation of the program.

Again, it is instructive to consider the division of labor between the BarChartCreator and BarChart classes. The chart class manages the bars, and it knows about the current selection state. The chart creator class has no knowledge of these details. It simply collects the user input and passes it to the chart.

worked_example_2/BarChartCreator.java

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.input.KeyCode;
import javafx.scene.layout.Pane;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class BarChartCreator extends Application
{
   private static final int DEFAULT_VALUE = 100;

   public void start(Stage primaryStage)
   {
      Pane root = createRootPane();
      Scene scene1 = new Scene(root);
      primaryStage.setScene(scene1);
      primaryStage.setTitle("BarChartCreator");
      primaryStage.show();
   }

   private Pane createRootPane()
   {
      TextField labelField = new TextField("");
      Button addButton = new Button("Add");
      Button removeButton = new Button("Remove");
      BarChart chart = new BarChart();
      addButton.setOnAction(event ->
         {
            chart.append(labelField.getText(), DEFAULT_VALUE);
            chart.requestFocus();
         });
      removeButton.setOnAction(event ->
         {
            chart.remove();
         });

      chart.setFocusTraversable(true);
      chart.setOnMousePressed(event ->
         {
            chart.selectBarAt(event.getY());
            chart.requestFocus();
         });

      chart.setOnMouseDragged(event ->
         {
            chart.setSelectedSize(event.getX());
         });

      chart.setOnKeyPressed(event -> {
         if (event.getCode() == KeyCode.RIGHT)
         {
            chart.growSelectedSize(1);
            event.consume();
         }
         else if (event.getCode() == KeyCode.LEFT)
         {
            chart.growSelectedSize(-1);
            event.consume();
         }
         else if (event.getCode() == KeyCode.UP)
         {
            chart.moveSelection(-1);
            event.consume();
         }
         else if (event.getCode() == KeyCode.DOWN)
         {
            chart.moveSelection(1);
            event.consume();
         }
      });

      chart.focusedProperty().addListener(obs ->
         {
            if (chart.isFocused())
            {
               chart.setStyle("-fx-border-style: dotted;"
                  + " -fx-border-width: 1px;"
                  + " -fx-border-color: blue;");
            }
            else
            {
               chart.setStyle("-fx-border-style: none");
            }
         });

      Pane pane1 = new VBox(10,
            new Label("Label:"),
            labelField,
            addButton,
            removeButton,
            chart);
      pane1.setPadding(new Insets(10));
      return pane1;
   }
}
```

worked_example_2/BarChart.java

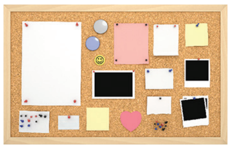
```
import javafx.scene.layout.Pane;

public class BarChart extends Pane
{
   private int selectedIndex;

   private static final int PANE_WIDTH = 400;
   private static final int PANE_HEIGHT = 400;
   private static final int GAP = 5;

   public BarChart()
   {
      selectedIndex = -1;
      setMinSize(PANE_WIDTH, PANE_HEIGHT);
   }

   private Bar getSelectedBar()
   {
      if (selectedIndex >= 0)
      {
         return (Bar) getChildren().get(selectedIndex);
      }
      else
      {
         return null;
      }
   }

   private void setSelectedIndex(int value)
   {
      int bars = getChildren().size();
      if (0 <= selectedIndex && selectedIndex < bars)
      {
         getSelectedBar().setSelected(false);
      }

      selectedIndex = value;
      if (selectedIndex < -1) { selectedIndex = -1; }
      else if (selectedIndex >= bars) { selectedIndex = bars - 1; }

      if (selectedIndex != -1)
      {
         getSelectedBar().setSelected(true);
      }
   }

   public void selectBarAt(double y)
   {
      setSelectedIndex((int) (y / (Bar.HEIGHT + GAP)));
   }

   public void moveSelection(int by)
   {
      setSelectedIndex(selectedIndex + by);
   }

   public void setSelectedSize(double x)
   {
      if (selectedIndex >= 0)
      {
         getSelectedBar().setSize(x);
      }
   }

   public void growSelectedSize(int dx)
   {
      if (selectedIndex >= 0)
      {
         getSelectedBar().growSize(dx);
      }
   }

   public void append(String label, double value)
   {
      Bar bar1 = new Bar(label, value);
      int bars = getChildren().size();
      bar1.relocate(0, (Bar.HEIGHT + GAP) * bars);
      getChildren().add(bar1);
      setSelectedIndex(bars);
   }

   public void remove()
   {
      if (selectedIndex >= 0)
      {
            getChildren().remove(selectedIndex);

            int bars = getChildren().size();
            for (int i = selectedIndex; i < bars; i++)
            {
               getChildren().get(i).relocate(0, (Bar.HEIGHT + GAP) * i);
            }
            setSelectedIndex(selectedIndex);
         }
      }
}
```

worked_example_2/Bar.java

```
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Rectangle;
import javafx.scene.text.Text;

public class Bar extends Pane
{
   public static final int HEIGHT = 15;
   private Rectangle rect1;

   public Bar(String label, double initialSize)
   {
      rect1 = new Rectangle(0, 0, initialSize, HEIGHT);

      Text text1 = new Text(label);
      text1.relocate(0, 0);
      text1.setStroke(Color.WHITE);

      getChildren().addAll(rect1, text1);
   }

   public void setSize(double x)
   {
      rect1.setWidth(x);
   }

   public void growSize(double dx)
   {
      rect1.setWidth(rect1.getWidth() + dx);
   }

   public void setSelected(boolean selected)
   {
      if (selected)
      {
         rect1.setFill(Color.RED);
      }
      else
      {
         rect1.setFill(Color.BLACK);
      }
   }
}
```

CHAPTER SUMMARY

Learn how to arrange multiple controls in a container.



- In JavaFX, you use layout panes to arrange user-interface controls.
- You can arrange controls by placing them in nested `HBox` and `VBox` panes.
- Use a `GridPane` to lay out controls in a grid of rows and columns.

Select among the JavaFX controls for presenting choices to the user.

- For a small set of mutually exclusive choices, use a group of radio buttons or a choice box.
- Add radio buttons to a `ToggleGroup` so that only one button in the group is selected at any time.
- For a binary choice, use a check box.
- For a large set of choices, use a choice box.
- Radio buttons, check boxes, and choice boxes generate action events, just as buttons do.

Implement menus in a JavaFX program.

- A menu bar contains menus. A menu contains submenus and menu items.
- A menu provides a list of available choices.
- Menu items generate action events.

Use properties and bindings.

- A property is accessed by getter and setter methods.
- Each property has a name and a type.
- An observable property notifies its listeners when its value changes.
- Attach property change handlers to `Property` objects.
- When a property is bound to another, it tracks the changes of the other property.

Implement animations in JavaFX.

- Key frames specify actions whose intermediate steps can be interpolated.
- A timeline plays key frames in parallel.
- Use a `SequentialTransition` to play one timeline after another.

Write programs that process mouse events.

- You use a mouse event handler to capture mouse events.

STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

```
javafx.animation.Animation
   play
   stop
   setCycleCount
   setAutoReverse
javafx.animation.KeyFrame
javafx.animation.KeyValue
javafx.animation.Timeline
```
`javafx.beans.Observable`
```
   addListener
```
`javafx.beans.Property`
```
   bind
   get
   set
   unbind
javafx.beans.binding.Bindings
   createObjectBinding
```
`javafx.beans.binding.NumberExpression`
```
   asString
javafx.beans.property.BooleanProperty
javafx.beans.property.DoubleProperty
javafx.beans.property.IntegerProperty
javafx.beans.property.ObjectProperty
javafx.beans.property.SimpleDoubleProperty
javafx.beans.property.StringProperty
javafx.fxml.FXMLLoader
   load
```
`javafx.fxml.Initializable`
```
   initialize
javafx.scene.Node
   focusedProperty
   requestFocus
   setFocusTraverable
   setId
   setOnKeyPressed
   setOnMouseDragged
   setOnMouseEntered
   setOnMouseExited
   setOnMouseMoved
   setOnMousePressed
   setOnMouseReleased
   setStyle
javafx.scene.chart.NumberAxis
   setLabel
javafx.scene.chart.LineChart
javafx.scene.chart.XYChart.Series
   getData
   setName
javafx.scene.control.ChoiceBox
   getItems
   getSelectionModel
javafx.scene.control.Menu
   getMenus
javafx.scene.control.MenuBar
javafx.scene.control.MenuItem
   getItems
javafx.scene.control.RadioButton
   isSelected
   setSelected
   setToggleGroup
javafx.scene.control.SelectionModel
   getSelectedItem
   select
javafx.scene.control.ToggleGroup
javafx.scene.input.KeyEvent
   getKeyCode
javafx.scene.input.KeyCode
   UP
   DOWN
   LEFT
   RIGHT
javafx.scene.input.MouseEvent
   getX
   getY
javafx.scene.layout.ColumnConstraints
   setPercentWidth
javafx.scene.layout.GridPane
   add
   getColumnConstraints
   setHalignment
   setHgap
   setPadding
   setVgap
javafx.scene.layout.HBox
javafx.util.Duration
   seconds
   millis
```

REVIEW EXERCISES

■ **R11.1** What is the difference between the gaps and the padding in an `HBox` or `VBox`?

■ **R11.2** What is the advantage of using nested boxes or a grid pane over using the `relocate` method that places controls at specific pixel positions?

■■■ **R11.3** Consider the last pane in Section 11.1.1. How can you center the label?

■■ **R11.4** What happens when you place two buttons in the same cell of a `GridPane`? Try it out by writing a sample program that adds two buttons of different sizes.

■■ **R11.5** What happens if you place a text field into the (0, 0) cell of a grid pane and make it span multiple rows, and then you place another text field into the (0, 0) cell that spans multiple columns? Try it out by writing a sample program.

■ **R11.6** Can you add a grid pane to a grid pane? Why why not?

■■ **R11.7** Can you add a button to a button? Why or why not?

■■ **R11.8** Build the last pane in Section 11.1.1 with the SceneBuilder and describe the steps that you carried out.

■ **R11.9** What is the difference between radio buttons and check boxes?

■ **R11.10** Why do you need a toggle group for radio buttons but not for check boxes?

■■■ **R11.11** Look up the JavaFX documentation for the `BarChart` class. How can you change the program in Programming Tip 11.2 to show a bar chart?

■■■ **R11.12** Suppose that we want to draw a box around the "Log base" label and the three radio buttons in Worked Example 11.1. How can you achieve that? (*Hint:* Special Topic 11.1.)

■ **R11.13** What is the difference between a menu bar, a menu, and a menu item?

■■ **R11.14** List the names and types of five properties of the `Circle` class, including one whose type is `String` and one whose type is some other class.

■ **R11.15** Is `prefSize` a property of the `Region` class? Why or why not?

■ **R11.16** Is `border` a property of the `Region` class? Why or why not?

■■ **R11.17** Is `userData` a property of the `Menu` class? Is it observable? Why or why not?

■■■ **R11.18** Suppose you want to change the font size of a label with a slider. Show how to accomplish that task by using a listener and by using a binding.

■ **R11.19** How can you show an animation of a rectangle that gets longer and longer, until it fills the width of the entire pane?

■ **R11.20** How can you show an animation of two balls that move toward each other until they touch?

■■ **R11.21** How can you show an animation of a ball that moves to trace the sides of a square?

■■ **R11.22** Suppose you want to show an animation of a swinging pendulum. Why can't you use key frames and key values?

■ **R11.23** What is the difference between an `ActionEvent` and a `MouseEvent`?

■■ **R11.24** How can you write a program that allows a user to click on two points, then draws the line segment joining them?

PRACTICE EXERCISES

■ **E11.1** Write an application with three buttons labeled "Red", "Green", and "Blue" that changes the background color of a pane to red, green, or blue.

■■ **E11.2** Add icons to the buttons of Exercise • E11.1. Consult the JavaFX API documentation for adding a "graphic".

- **• E11.3** Write a program that displays a login dialog with labeled text fields for the user name and password, a login button, and a label for messages. Simply display the message "Invalid attempt" whenever the button is pressed.
- **• E11.4** Write an application with three radio buttons labeled "Red", "Green", and "Blue" that changes the background color of a pane to red, green, or blue.
- **• E11.5** Write an application with three check boxes labeled "Red", "Green", and "Blue" that adds a red, green, or blue component to the background color of a pane. This application can display a total of eight color combinations.
- **• E11.6** Write an application with a choice box containing three items labeled "Red", "Green", and "Blue" that change the background color of a pane in the center of the stage to red, green, or blue.
- **• E11.7** Write an application with a Color menu and menu items labeled "Red", "Green", and "Blue" that change the background color of a pane in the center of the stage to red, green, or blue.
- **• E11.8** Write a program that displays a number of rectangles at random positions. Supply menu items "Fewer" and "More" that generate fewer or more random rectangles. Each time the user selects "Fewer", the count should be halved. Each time the user selects "More", the count should be doubled.
- **•• E11.9** Modify the program of Exercise • E11.8 to replace the buttons with a slider for generating more or fewer random rectangles.
- **•• E11.10** Write a program with a horizontal and a vertical slider that can move a circle anywhere inside a pane.
- **• E11.11** Add labels to the color viewer of Section 11.4 that show the color values.
- **••• E11.12** Write a program that simulates a signup form with fields for first name, last name, and user name. By default, the user name consists of the first letter of the first name, followed by the last name, in lowercase. For example, if the first and last name are Joanne Smith, the user name is `jsmith`. Update the user name field as the contents of the first and last name fields change. Also allow the user to change the user name explicitly. *Hint:* `bind`, `unbind`.
- **••• E11.13** Write a program that simulates an order form with shipping and billing addresses. When a check box "same as shipping" is clicked, any changes to the shipping address automatically update the billing address. *Hint:* `bind`, `unbind`.
- **•• E11.14** Write a program that simulates the volume sliders for the left and right speakers of a stereo player. When a check box "mono" is clicked, adjusting either slider should set the other one to the same value. *Hint:* `bind`, `unbind`.
- **• E11.15** Write a program that makes a circle gradually fade away as it moves toward the right of a pane. *Hint:* Change the opacity.
- **•• E11.16** Write a program that makes a ball bounce up and down indefinitely.
- **••• E11.17** When a real ball bounces up and down, it loses a bit of energy each time. Write a program that simulates this, by reducing the maximum height by 5 percent in each bounce.
- **• E11.18** Write a program that makes a car move along the screen. Use an image.

■ **E11.19** Write a program that animates two cars moving across a pane in opposite directions (but at different heights so that they don't collide).

■■ **E11.20** Write a program that makes two cars move toward each other, until they meet. Then make them reverse.

■■ **E11.21** Write a program that simulates a digital clock, showing the current time once per second. You can get the current time by calling `Instant.now().toString()`. The `Instant` class is in the `java.time` package.

■■ **E11.22** Change the `MouseDemo` program in Section 11.6 so that a new circle is added to the pane whenever the mouse is clicked on an empty area.

■■ **E11.23** Write a program with a button "Click me" that moves away as the mouse approaches it.

■ **E11.24** Write a program that prompts the user to enter the *x*- and *y*-positions of a center point and a radius, using text fields. When the user clicks a "Draw" button, draw a circle with that center and radius in a component.

■ **E11.25** Write a program that allows the user to specify a circle by typing the radius in a text field and then clicking on the center. Note that you don't need a "Draw" button.

■ **E11.26** Write a program that allows the user to specify a circle with two mouse presses: the first one indicates the center point and the second one a point on the periphery. *Hint:* In the mouse press handler, you must keep track of whether you already received the center point in a previous mouse press.

■■ **E11.27** Write a program that allows the user to specify a triangle with three mouse presses. After the first mouse press, draw a small dot. After the second mouse press, draw a line joining the first two points. After the third mouse press, draw the entire triangle. The fourth mouse press erases the old triangle and starts a new one.

■■■ **E11.28** Write a program that allows the user to specify a circle with three mouse presses. After the first mouse press, draw a small dot. After the second mouse press, draw a second dot. After the third mouse press, draw the circle that passes through the three points. The fourth mouse press erases the old circle and starts a new one.

■ **E11.29** In the program of Worked Example 11.2, allow users to press the "Delete" key on the keyboard to remove the currently selected bar.

PROGRAMMING PROJECTS

■■ **P11.1** Enhance the font viewer program to allow the user to select different font faces. Research the API documentation to find out how to locate the available fonts on the user's system.

■■ **P11.2** Add a `width` property to the `ItalianFlag` class of Worked Example 10.1. Write a program with a slider that changes the width of a flag.

■■■ **P11.3** Make a class `StripedFlag` with `Color`-valued properties `color1`, `color2`, and `color3`, and a `boolean`-valued property `horizontal`. Provide a user interface for setting the colors and the orientation, and for updating the flag according to the user choices.

■■■ **P11.4** Implement a `TextLabel` class with properties for the font name and font size, as well as `boolean`-valued properties `italic` and `bold`. Use that class to reimplement the program of Section 11.2, but use a slider for the font size.

■■■ **P11.5** Write a program that models a ball bouncing in a rectangle. When the ball meets the rectangle boundary, it should bounce properly, so that the incoming and outgoing angle are the same.

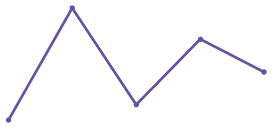
■■■ **P11.6** Write a program that models the motion of an object that is attached to a spring. At time t, the displacement is $m \sin(f\,t)$, where m is the maximum displacement and f depends on the stiffness of the spring.

■■■ **P11.7** Write a program that displays a scrolling message in a pane. When the message has left the window, reset the starting position to the other corner. Provide a user interface to customize the message text, font, foreground and background colors, and the scrolling speed.

■■■ **P11.8** Implement a program that allows two players to play tic-tac-toe. Draw the game grid and an indication of whose turn it is (X or O). Upon the next click, check that the mouse click falls into an empty location, fill the location with the mark of the current player, and give the other player a turn. If the game is won, indicate the winner. Also supply a button for starting over.

© Kathy Muller/iStockphoto.

■■ **P11.9** Write a program that lets users design charts such as the following:



Use appropriate controls to ask for the x- and y-positions of the points, and to redraw the chart when the user adds an item.

■■ **P11.10** Write a program that lets users design line charts with a mouse. When the user drags an existing point, the point is moved. (Allow for a few pixels of tolerance.) When the user clicks elsewhere, a point is added to the chart. When the user clicks on an existing point, and then clicks a “Delete” button, the point is removed.

■■ **P11.11** Write a program that lets users design pie charts, using the JavaFX pie chart control. Provide a text area for the data points and a “Draw” button that draws the chart.

■■ **Business P11.12** Write a program with a graphical interface that allows the user to convert an amount of money between U.S. dollars (USD), Japanese yen (JPY), euros (EUR), and British pounds (GBP). The user interface should have the following elements: a text box to enter the amount to be converted, two choice boxes to allow the user to select the currencies, a button to make the conversion, and a label to show the result. Display a warning if the user does not choose different currencies.

You can get up-to-date exchange rates from `http://www.ecb.europa.eu/stats/exchange/eurofxref/html/index.en.html`, or download them from `http://www.ecb.europa.eu/stats/eurofxref/eurofxref-daily.xml` in a format that is easier to parse.

■■ Business P11.13 Write a program with a graphical interface that implements a login window with text fields for the user name and password. When the login is successful, hide the login window and open a new window with a welcome message. Follow these rules for validating the password:

- The user name is not case sensitive.
- The password is case sensitive.
- The user has three opportunities to enter valid credentials.

Otherwise, display an error message and terminate the program. When the program starts, read the file `users.txt`. Each line in that file contains a user name and password, separated by a space.

ANSWERS TO SELF-CHECK QUESTIONS

1. Place them inside a `VBox`, not an `HBox`. You can just put them into the same `VBox` that contains the label, as we did in the preceding chapter.

2. Use a `VBox` that contains:

- A `TextField`
- An `HBox` holding the buttons 7, 8, 9
- Three more `HBox` panes, each holding three buttons.

Unfortunately, the buttons won't line up perfectly because the "." button is smaller than the others, and the "CE" button is larger.

3. The second button is placed on top of the first one.

4. `pane.add(button0, 0, 4, 2, 1);`

5. In the first and third row of the `GridPane`, add a single control and have it span three columns.

6. If you have many options, a set of radio buttons takes up a large area. A choice box can show many options without using up much space. But the user cannot see the options as easily.

7. If one of them is checked, the other one is unchecked. You should use radio buttons if that is the behavior you want.

8. Instead of using radio buttons with two choices, use a check box.

9. When any of the component settings is changed, the program simply queries all of them and updates the label.

10. Simply call

```
fontChoice.getItems().addAll(
   Font.getFamilies());
```

No other change is necessary.

11. When you open a menu, you have not yet made a selection. Only `MenuItem` objects correspond to selections.

12. The compiler reports an error. You can only add menus to a menu bar.

13. The program will compile, but there will be a warning when the program runs, and the child menu will not actually be added.

14. Then the `faceName` variable is set when the menu item is added to the menu, not when the user selects the menu.

15. In the previous program, the user-interface components effectively served as storage for the font specification. Their current settings were used to construct the font. But a menu doesn't save settings; it just generates an action.

16. No. There is a `getSize` method, but not a `setSize` method.

17. Yes, `font` is a property. There are getter and setter methods

```
Font getFont()
void setFont(Font value)
```

It is an observable property. The method `fontProperty()` yields a property object to which you can attach a listener.

18. Action events describe one-time changes, such as button clicks. Sliders emit property change events whenever the slider position changes.

19. Call

```
ball.radiusProperty().bind(
   positionSlider.valueProperty());
```

20. Now the circle's `fill` property depends on the three `text` properties. In the computation, convert strings to integers.

```
ball.fillProperty().bind(
   Bindings.createObjectBinding(
      () -> Color.rgb(
         Integer.parseInt(redSlider.getText()),
         Integer.parseInt(
            greenSlider.getText()),
         Integer.parseInt(
            blueSlider.getText())),
       redField.textProperty(),
       greenField.textProperty(),
       blueField.textProperty()));
```

21. Construct a circle with radius 100, then play a timeline that grows the radius to the desired value:

```
KeyFrame frame1 = new KeyFrame(
   Duration.seconds(2),
   new KeyValue(ball.radiusProperty(), 200));
Timeline animation = new Timeline(frame1);
```

22. Construct a rectangle with width and height 100, then play a timeline that grows both the width and the height:

```
KeyFrame frame1 = new KeyFrame(
   Duration.seconds(2),
   new KeyValue(rect.widthProperty(), 200),
   new KeyValue(rect.heightProperty(), 200));
Timeline animation = new Timeline(frame1);
```

23. Join two timelines by a sequential transition:

```
KeyFrame frame1 = new KeyFrame(
   Duration.seconds(2),
   new KeyValue(ball.radiusProperty(), 200));
KeyFrame frame2 = new KeyFrame(
   Duration.seconds(4),
   new KeyValue(ball.radiusProperty(), 0));
SequentialTransition animation =
   new SequentialTransition(
      new Timeline(frame1),
      new Timeline(frame2));
```

24. Set the timeline to cycle and autoreverse, like this:

```
KeyFrame frame1 = new KeyFrame(
   Duration.seconds(1),
   new KeyValue(ball.radiusProperty(), 110));
Timeline timeline1 = new Timeline(frame1);
animation.setCycleCount(Animation.INDEFINITE);
animation.setAutoReverse(true);
```

25. Construct a circle whose `centerY` is initially `y0`. Make a key frame with a short duration that calls a handler upon completion to update the ball position, like this:

```
long start = System.currentTimeMillis();
Circle ball = new Circle(radius, y0, radius);
KeyFrame frame1 = new KeyFrame(
   Duration.millis(10),
   e ->
      {
         double t =
            System.currentTimeMillis() - start;
         ball.setCenterY(y0 + m
            * Math.sin(f * t));
      });
```

Then repeat the timeline indefinitely and play it.

26. Provide a handler that adds the circle whose center is the mouse position:

```
pane.setOnMousePressed(event ->
   pane.getChildren().add(new Circle(
   event.getX(), event.getY(), radius)));
```

27. Add handlers to the circle, like this:

```
ball.setOnMouseEntered(
   event -> ball.setFill(Color.YELLOW));
ball.setOnMouseExited(
   event -> ball.setFill(Color.BLACK));
```

28. First, provide a square:

```
Rectangle square =
   new Rectangle(75, 25, 50, 50);
```

The `Rectangle` class doesn't have `centerX` and `centerY` properties. If you want the center of the square to be at the mouse pointer position, you need to make an adjustment:

```
root.setOnMousePressed(event ->
   {
      square.setX(event.getX()
         - square. getWidth() / 2);
      square.setY(event.getY()
         - square. getHeight() / 2);
   });
```

29. If you press the mouse button inside the circle, the program works as before. But if you press the mouse button outside the circle and drag the mouse, the circle will not follow.

30. There is no difference in behavior. The handler turns the circle back to black when dragging is finished. At that time, the mouse pointer will be inside the circle.