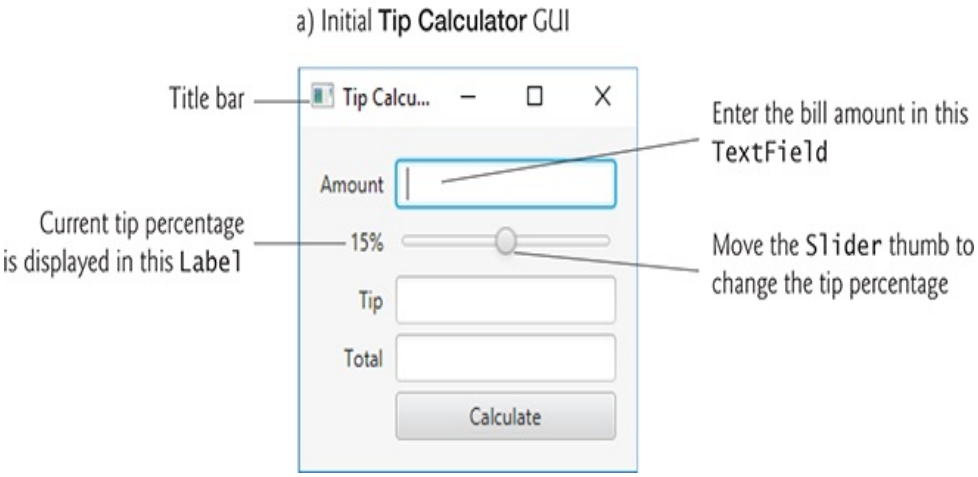
12.5 Tip Calculator App—Introduction to Event Handling

The **Tip Calculator** app (Fig. 12.8(a)) calculates and displays a restaurant bill tip and total. By default, the app calculates the total with a 15% tip. You can specify a tip percentage from 0% to 30% by moving the `Slider` *thumb*—this updates the tip percentage (Fig. 12.8(b)) and (c)). In this section, you'll build a **Tip Calculator** app using several JavaFX components and learn how to respond to user interactions with the GUI.



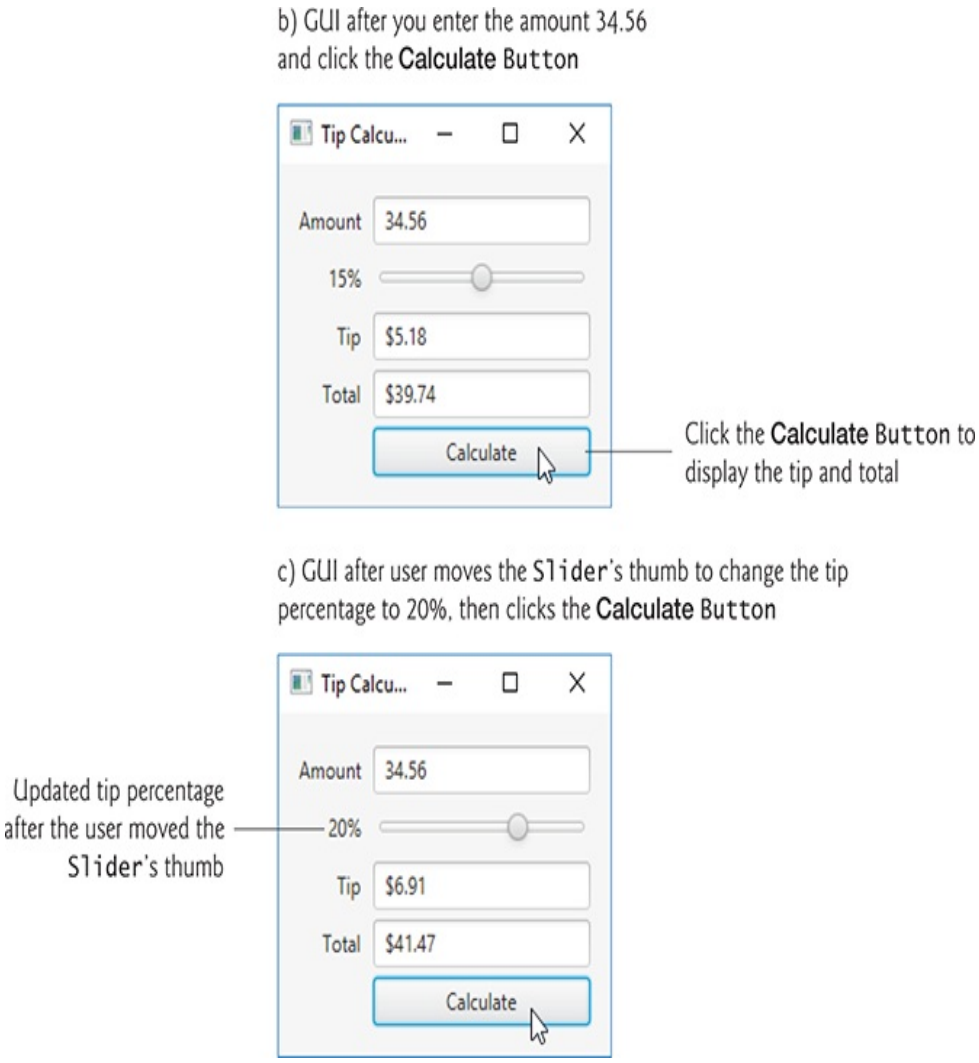


Fig. 12.8

Entering the bill amount and calculating the tip.

Description

You'll begin by test-driving the app, using it to calculate 15% and 20% tips. Then we'll overview the technologies you'll use to create the app. You'll build the app's GUI using the Scene Builder. Finally, we'll present the complete Java code for the

app and do a detailed code walkthrough.

12.5.1 Test-Driving the Tip Calculator App

Compile and run the app located in the `TipCalculator` folder with this chapter's examples. The class containing the `main` method is named `TipCalculator`.

Entering a Bill Total

Using your keyboard, enter `34.56`, then press the **Calculate** `Button`. The **Tip** and **Total** `TextField`s show the tip amount and the total bill for a 15% tip (Fig. 12.8(b)).

Selecting a Custom Tip Percentage

Use the `Slider` to specify a *custom* tip percentage. Drag the `Slider`'s *thumb* until the percentage reads **20%** (Fig. 12.8(c)), then press the **Calculate** `Button` to display the updated tip and total. As you drag the thumb, the tip percentage in the `Label` to the `Slider`'s left updates continuously. By default, the `Slider` allows you to select values from 0.0 to 100.0, but in this app we'll restrict the

`Slider` to selecting whole numbers from 0 to 30.

12.5.2 Technologies Overview

This section introduces the technologies you'll use to build the **Tip Calculator** app.

Class `Application`

The class responsible for launching a JavaFX app is a subclass of `Application` (package `javafx.application`). When the subclass's `main` method is called:

1. Method `main` calls class `Application`'s `static` `launch` method to begin executing the app.
2. The `launch` method, in turn, causes the JavaFX runtime to create an object of the `Application` subclass and call its `start` method.
3. The `Application` subclass's `start` method creates the GUI, attaches it to a `Scene` and places it on the `Stage` that `start` receives as an argument.

Arranging JavaFX Components with a `GridPane`

Recall that layout containers arrange JavaFX components in a `Scene`. A `GridPane` (package `javafx.scene.layout`) arranges JavaFX components into *columns* and *rows* in a rectangular grid.

This app uses a `GridPane` (Fig. 12.9) to arrange views into two columns and five rows. Each cell in a `GridPane` can be empty or can hold one or more JavaFX components, including layout containers that arrange other controls. Each component in a `GridPane` can span *multiple* columns or rows, though we did not use that capability in this GUI. When you drag a `GridPane` onto Scene Builder's content panel, Scene Builder creates the `GridPane` with two columns and three rows by default. You can add and remove columns and rows as necessary. We'll discuss other `GridPane` features as we present the GUI-building steps. To learn more about class `GridPane`, visit:

```
https://docs.oracle.com/javase/8/javafx/api/javafx/sc
```

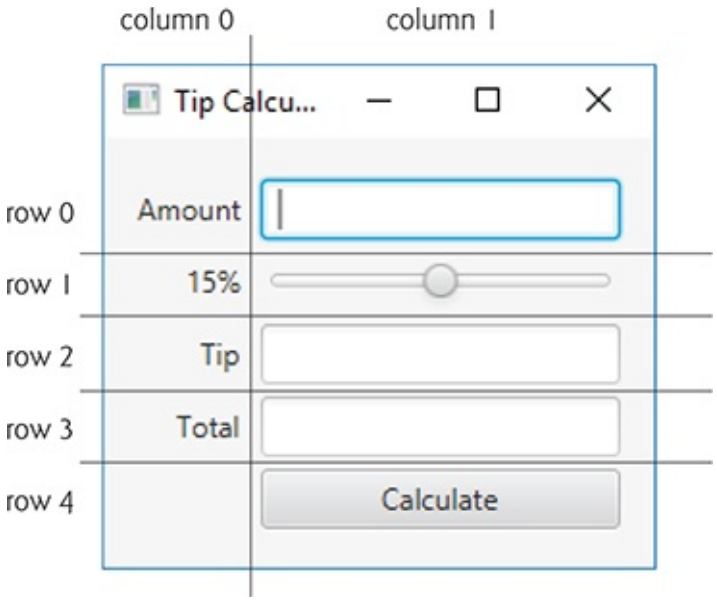


Fig. 12.9

Tip Calculator GUI's `GridPane` labeled by its rows and columns.

Description

Creating and Customizing the GUI with Scene Builder

You'll create `Label`s, `TextField`s, a `Slider` and a `Button` by dragging them onto Scene Builder's content panel, then customize them using the **Inspector** window.

- A `TextField` (package `javafx.scene.control`) can accept text input from the user or display text. You'll use one editable `TextField` to input the bill amount from the user and two *uneditable* `TextField`s to

display the tip and total amounts.

- A `Slider` (package `javafx.scene.control`) represents a value in the range 0.0–100.0 by default and allows the user to select a number in that range by moving the `Slider`'s thumb. You'll customize the `Slider` so the user can choose a custom tip percentage *only* from the more limited integer range 0 to 30.
- A `Button` (package `javafx.scene.control`) allows the user to initiate an action—in this app, pressing the **Calculate** `Button` calculates and displays the tip and total amounts.

Formatting Numbers as Locale-Specific Currency and Percentage Strings

You'll use class `NumberFormat` (package `java.text`) to create *locale-specific* currency and percentage strings—an important part of *internationalization*.[2]

2. Recall that the new JavaMoney API (`http://javamoney.github.io`) was developed to meet the challenges of handling currencies, monetary amounts, conversions, rounding and formatting. At the time of this writing, it was not yet incorporated into the JDK.

Event Handling

Normally, a user interacts with an app's GUI to indicate the tasks that the app should perform. For example, when you write an e-mail, clicking the e-mail app's **Send** button tells the app to send the e-mail to the specified e-mail addresses.

GUIs are **event driven**. When the user interacts with a GUI component, the interaction—known as an **event**—drives the program to perform a task. Some common user interactions that cause an app to perform a task include *clicking* a button, *typing* in a text field, *selecting* an item from a menu, *closing* a window and *moving* the mouse. The code that performs a task in response to an event is called an **event handler**, and the process of responding to events is known as **event handling**.

Before an app can respond to an event for a particular control, you must:

1. Define an event handler that implements an appropriate interface—known as an **event-listener interface**.
2. Indicate that an object of that class should be notified when the event occurs—known as **registering the event handler**.

In this app, you'll respond to two events—when the user moves the `Slider`'s thumb, the app will update the `Label` that displays the current tip percentage, and when the user clicks the **Calculate** `Button`, the app will calculate and display the tip and total bill amount.

You'll see that for certain events—such as when the user clicks a `Button`—you can link a control to its event-handling method by using the **Code** section of Scene Builder's **Inspector** window. In this case, the event-listener interface is implemented for you to call the method that you specify. For events that occur when the value of a control's property changes—such as when the user moves a `Slider`'s thumb to change the `Slider`'s value—you'll see that you must create

the event handler entirely in code.

Implementing Interface `ChangeListener` for Handling `Slider` Thumb Position Changes

You'll implement interface `ChangeListener` (package `javafx.beans.value`) to respond when the user moves the `Slider`'s thumb. In particular, you'll use the interface's `changed` method to display the updated tip percentage as the user moves the `Slider`'s thumb.

Model-View-Controller (MVC) Architecture

JavaFX applications in which the GUI is implemented as FXML adhere to the **Model-View-Controller (MVC) design pattern**, which separates an app's data (contained in the **model**) from the app's GUI (the **view**) and the app's processing logic (the **controller**).

The controller implements logic for processing user inputs. The model contains application data, and the view presents the data stored in the model. When a user provides some input, the

controller modifies the model with the given input. In the **Tip Calculator**, the model is the bill amount, the tip and the total. When the model changes, the controller updates the view to present the changed data.

In a JavaFX FXML app, a **controller class** defines instance variables for interacting with controls programmatically, as well as event-handling methods that respond to the user's interactions. The controller class may also declare additional instance variables, `static` variables and methods that support the app's operation. In a simple app like the **Tip Calculator**, the model and controller are often combined into a single class, as we'll do in this example.

`FXMLLoader` Class

When a JavaFX FXML app begins executing, class `FXMLLoader`'s `static` method `load` is used to load the FXML file that represents the app's GUI. This method:

- Creates the GUI's scene graph—containing the GUI's layouts and controls—and returns a `Parent` (package `javafx.scene`) reference to the scene graph's root node.
- Initializes the controller's instance variables for the components that are manipulated programmatically.
- Creates and registers the event handlers for any events specified in the FXML.

We'll discuss these steps in more detail in Sections 12.5.4–12.5.5.

12.5.3 Building the App's GUI

In this section, we'll show the precise steps for creating the **Tip Calculator**'s GUI. The GUI will not look like the one shown in Fig. 12.8 until you've completed the steps.

fx:id Property Values for This App's Controls

If the controller class will manipulate a control or layout programmatically (as we'll do with one `Label`, all the `TextField`s and the `Slider`), you must provide a name for that control or layout. In Section 12.5.4, you'll learn how to declare Java variables for each such component in the FXML, and we'll discuss how those variables are initialized for you. Each object's name is specified via its **fx:id property**. You can set this property's value by selecting a component in your scene, then expanding the **Inspector** window's **Code** section —the **fx:id** property appears at the top of the **Code** section. Figure 12.10 shows the **fx:id** properties of the **Tip Calculator**'s programmatically manipulated controls. For clarity, our naming convention is to use the control's class name in the **fx:id** property.

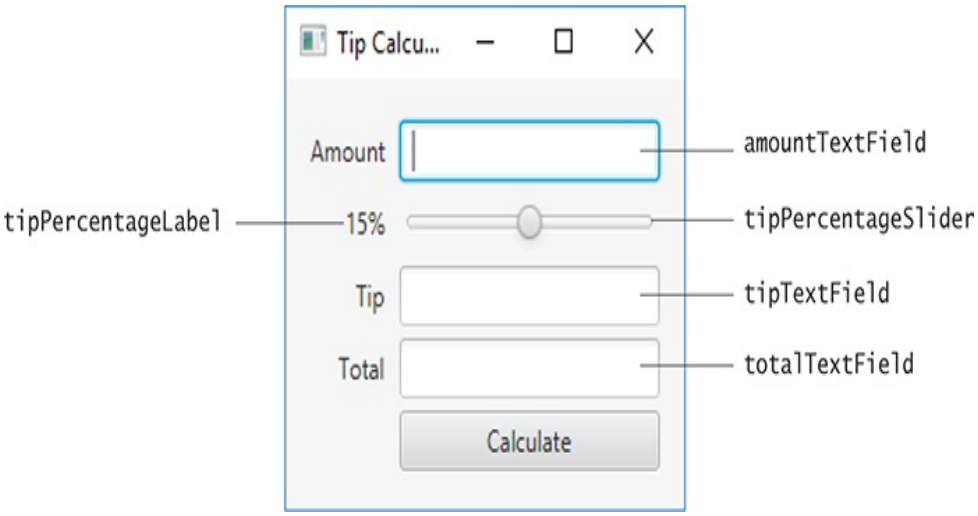


Fig. 12.10

Tip Calculator's programmatically manipulated controls labeled with their **fx:id**s.

Description

Creating the `TipCalculator.fxml` File

As you did in Section 12.4.1, open Scene Builder to create a new FXML file. Then, select **File** > **Save** to display the **Save As** dialog, specify the location in which to store the file, name the file `TipCalculator.fxml` and click the **Save** button.

Step 1: Adding a `GridPane`

Drag a `GridPane` from the **Library** window's **Containers** section onto Scene Builder's content panel. By default, the `GridPane` contains two columns and three rows as shown in Fig. 12.11.

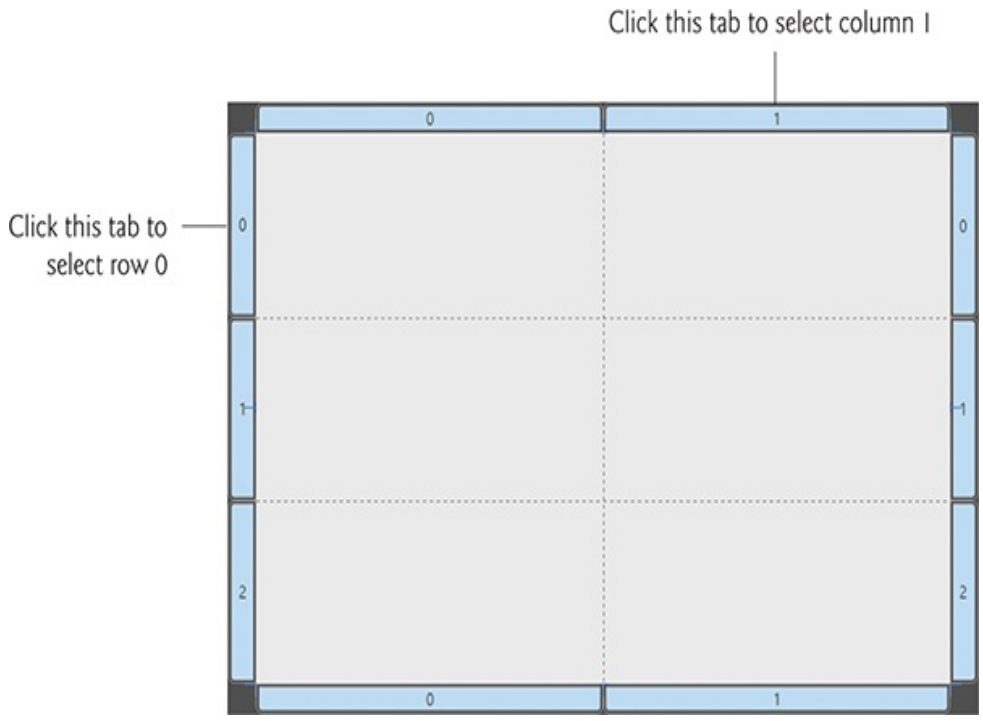


Fig. 12.11

`GridPane` with two columns (0 and 1) and three rows (0, 1 and 2).

Step 2: Adding Rows to the

GridPane

Recall that the GUI in Fig. 12.9 has two columns and five rows. Here you'll add two more rows. To add a row above or below an existing row:

1. Right click any row's row number tab and select either **Grid Pane > Add Row Above** or **Grid Pane > Add Row Below**.
2. Repeat this process to add another row.

After adding two rows, the `GridPane` should appear as shown in Fig. 12.12. You can use similar steps to add columns. You can delete a row or column by right clicking the tab containing its row or column number and selecting **Delete**.

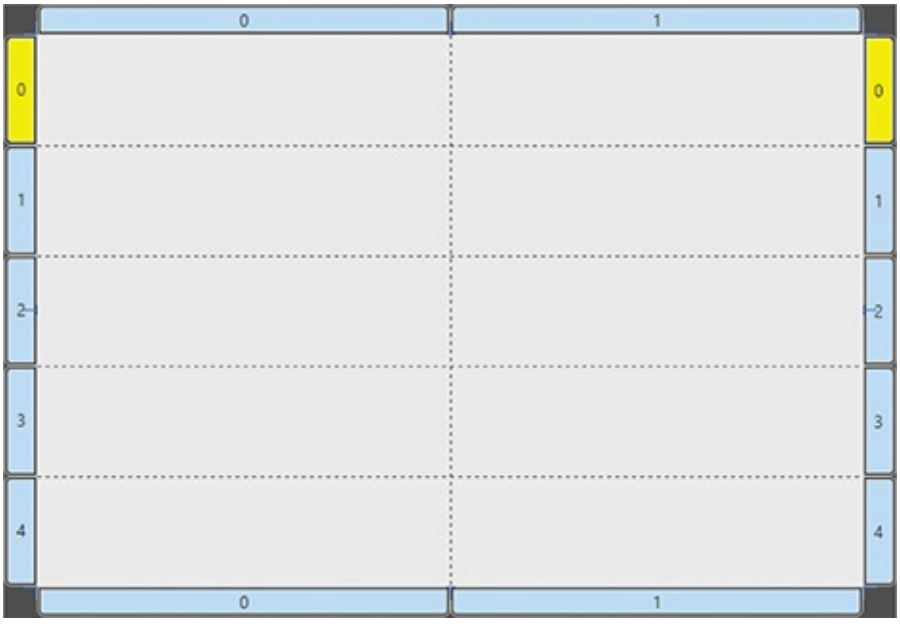


Fig. 12.12

`GridPane` after adding two more rows.

Step 3: Adding the Controls to the `GridPane`

You'll now add the controls in Fig. 12.9 to the `GridPane`. For each control that has an **fx:id** in Fig. 12.10, when you drag the control onto the `GridPane`, set the control's **fx:id** property in the **Inspector** window's **Code** section. Perform the following steps:

1. ***Adding the*** `Label`***s.*** Drag `Label`s from the **Library** window's **Controls** section into the first four rows of column 0 (the `GridPane`'s left column). As you add each `Label`, set its text as shown Fig. 12.9.
2. ***Adding the*** `TextField`***s.*** Drag `TextField`s from the **Library** window's **Controls** section into rows 0, 2 and 3 of column 1 (the `GridPane`'s right column).
3. ***Adding a*** `Slider`**.** Drag a horizontal `Slider` from the **Library** window's **Controls** section into row 1 of column 1.
4. ***Adding a*** `Button`**.** Drag a `Button` from the **Library** window's **Controls** section into row 4 of column 1. Change the `Button`'s text to **Calculate**. You can set the `Button`'s text by double clicking it, or by selecting the `Button`, then setting its **Text** property in the **Inspector** window's **Properties** section.

The `GridPane` should appear as shown in Fig. 12.13.

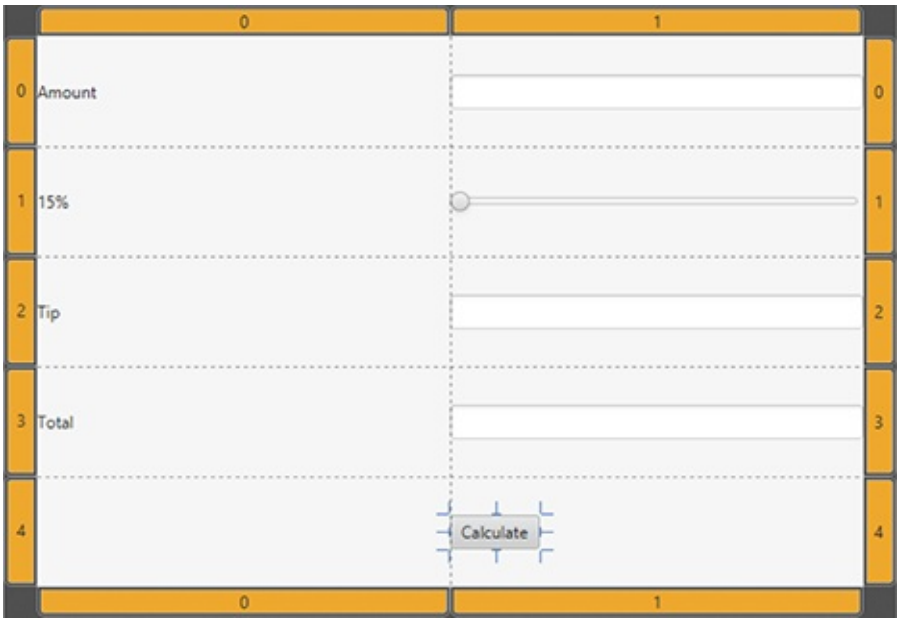


Fig. 12.13

`GridPane` filled with the **Tip Calculator**'s controls.

Description

Step 4: Sizing the `GridPane` to Fit Its Contents

When you begin designing a GUI by adding a layout, Scene Builder automatically sets the layout object's **Pref Width** property to `600` and **Pref Height** property to `400`, which is much larger than this GUI's final width and height. For this app, we'd like the layout's size to be computed, based on the

layout's contents. To make this change:

1. First, select the `GridPane` by clicking inside the `GridPane`, but not on any of the controls you've placed into its columns and rows. Sometimes, it's easier to select the `GridPane` node in the Scene Builder **Document** window's **Hierarchy** section.
2. In the **Inspector**'s **Layout** section, reset the **Pref Width** and **Pref Height** property values to their defaults (as you did in Section 12.4.4). This sets both properties' values to `USE_COMPUTED_SIZE`, so the layout calculates its own size.

The layout now appears as shown in Fig. 12.14.

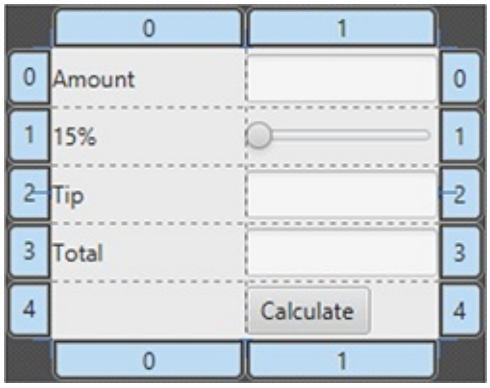


Fig. 12.14

`GridPane` sized to fit its contents.

Description

Step 5: Right-Aligning `GridPane` Column 0's

Contents

A `GridPane` column's contents are left-aligned by default. To right-align the contents of column 0, select it by clicking the tab at the top or bottom of the column, then in the **Inspector**'s **Layout** section, set the **Halignment** (horizontal alignment) property to `RIGHT`.

Step 6: Sizing the `GridPane` Columns to Fit Their Contents

By default, Scene Builder sets each `GridPane` column's width to 100 pixels and each row's height to 30 pixels to ensure that you can easily drag controls into the `GridPane`'s cells. In this app, we sized each column to fit its contents. To do so, select the column 0 by clicking the tab at the top or bottom of the column, then in the **Inspector**'s **Layout** section, reset the **Pref Width** property to its default size (that is, `USE_COMPUTED_SIZE`) to indicate that the column's width should be based on its widest child—the **Amount** `Label` in this case. Repeat this process for column 1. The `GridPane` should appear as shown in Fig. 12.15.

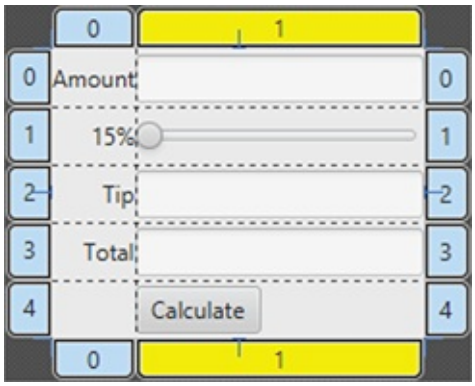


Fig. 12.15

`GridPane` with columns sized to fit their contents.

Description

Step 7: Sizing the `Button`

By default, Scene Builder sets a `Button`'s width based on its text. For this app, we chose to make the `Button` the same width as the other controls in the `GridPane`'s right column. To do so, select the `Button`, then in the **Inspector**'s **Layout** section, set the **Max Width** property to `MAX_VALUE`. This causes the `Button`'s width to grow to fill the column's width.

Previewing the GUI

Preview the GUI by selecting **Preview > Show Preview in Window**. As you can see in Fig. 12.16, there's no space

between the `Label`s in the left column and the controls in the right column. In addition, there's no space around the `GridPane`, because by default the `Stage` is sized to fit the `Scene`'s contents. Thus, many of the controls touch the window's borders. You'll fix these issues in the next step.

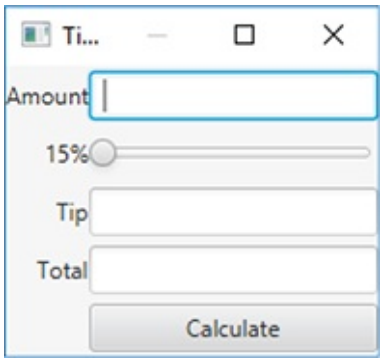


Fig. 12.16

`GridPane` with the `TextField`s and `Button` resized.

Description

Step 8: Configuring the `GridPane`'s Padding and Horizontal Gap Between Its Columns

The space between a node's contents and its top, right, bottom and left edges is known as the **padding**, which separates the

contents from the node's edges. Since the `GridPane`'s size determines the `Stage`'s window size, the `GridPane`'s padding separates its children from the window's edges. To set the padding, select the `GridPane`, then in the **Inspector**'s **Layout** section, set the **Padding** property's four values (which represent the **TOP**, **RIGHT**, **BOTTOM** and **LEFT**) to `14`—the JavaFX recommended distance between a control's edge and the `Scene`'s edge.

You can specify the default amount of space between a `GridPane`'s columns and rows with its **Hgap** (horizontal gap) and **Vgap** (vertical gap) properties, respectively. Because Scene Builder sets each `GridPane` row's height to 30 pixels—which is greater than the heights of this app's controls—there's already some vertical space between the components. To specify the horizontal gap between the columns, select the `GridPane` in the **Document** window's **Hierarchy** section, then in the **Inspector**'s **Layout** section, set the **Hgap** property to `8`—the recommended distance between controls. If you'd like to precisely control the vertical space between components, you can reset each row's **Pref Height** to its default value, then set the `GridPane`'s **Vgap** property.

Step 9: Making the `tipTextField` and `totalTextField` Uneditable and Not

Focusable

The `tipTextField` and `totalTextField` are used in this app only to display results, not receive text input. For this reason, they should not be interactive. You can type in a `TextField` only if it's "in **focus**"—that is, it's the control that the user is interacting with. When you click an interactive control, it receives the focus. Similarly, when you press the *Tab* key, the focus transfers from the current focusable control to the next one—this occurs in the order the controls were added to the GUI. Interactive controls—such as `TextField`s, `Slider`s and `Button`s—are focusable by default. Non-interactive controls—like `Label`s—are not focusable.

In this app, the `tipTextField` and `totalTextField` are neither editable nor focusable. To make these changes, select both `TextField`s, then in the **Inspector**'s **Properties** section uncheck the **Editable** and **Focus Traversable** properties. To select multiple controls at once, you can click the first (in the **Document** window's **Hierarchy** section or in the content panel), then hold the *Shift* key and click each of the others.

Step 10: Setting the `Slider`'s Properties

To complete the GUI, you'll now configure the **Tip**

Calculator's `Slider`. By default, a `Slider`'s range is `0.0` to `100.0` and its initial value is 0.0. This app allows only integer tip percentages in the range 0 to 30 with a default of 15. To make these changes, select the `Slider`, then in the **Inspector**'s **Properties** section, set the `Slider`'s **Max** property to `30` and the **Value** property to `15`. We also set the **Block Increment** property to `5`—this is the amount by which the **Value** property increases or decreases when the user clicks between an end of the `Slider` and the `Slider`'s thumb. Save the FXML file by selecting **File** > **Save**.

Though we set the **Max**, **Value** and **Block Increment** properties to integer values, the `Slider` still produces floating-point values as the user moves its thumb. In the app's Java code, we'll restrict the `Slider`'s values to integers when we respond to its events.

Previewing the Final Layout

You've now completed the **Tip Calculator**'s design. Select **Preview** > **Show Preview in Window** to view the final GUI (Fig. 12.17). When we discuss the `TipCalculatorController` class in Section 12.5.5, we'll show how to specify the **Calculate** `Button`'s event handler in the FXML file.

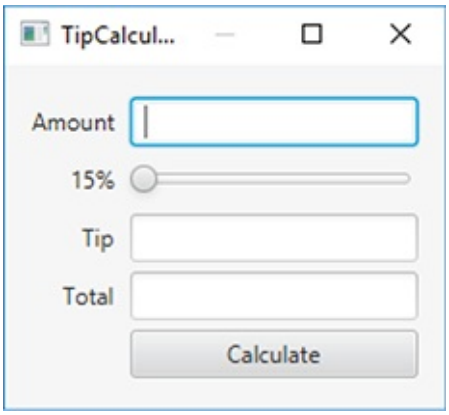


Fig. 12.17

Final GUI design previewed in Scene Builder.

Description

Specifying the Controller Class's Name

As we mentioned in Section 12.5.2, in a JavaFX FXML app, the app's controller class typically defines instance variables for interacting with controls programmatically, as well as event-handling methods. To ensure that an object of the controller class is created when the app loads the FXML file at runtime, you must specify the controller class's name in the FXML file:

1. Expand Scene Builder **Document** window's **Controller** section (located below the **Hierarchy** section in Fig. 12.3).
2. In the **Controller Class** field, type `TipCalculatorController`—by

convention, the controller class's name starts with the same name as the FXML file (`TipCalculator`) and ends with `Controller`.

Specifying the Calculate `Button`'s Event-Handler Method Name

You can specify in the FXML file the names of the methods that will be called to handle specific control's events. When you select a control, the **Inspector** window's **Code** section shows all the events for which you can specify event handlers in the FXML file. When the user clicks a `Button`, the method specified in the **On Action** field is called—this method is defined in the controller class you specify in Scene Builder's **Controller** window. Enter `calculateButtonPressed` in the **On Action** field.

Generating a Sample Controller Class

You can have Scene Builder generate the initial controller class containing the variables you'll use to interact with controls programmatically and the empty **Calculate** `Button` event handler. Scene Builder calls this the "controller skeleton." Select **View > Show Sample Controller Skeleton** to generate the skeleton (Fig. 12.18). As you can see, the

sample class has the class name you specified, a variable for each control that has an **fx:id** and an empty **Calculate** `Button` event handler. We'll discuss the `@FXML` annotation in Section 12.5.5 To use this skeleton to create your controller class, you can click the **Copy** button, then paste the contents into a file named `TipCalculatorController.java` in the same folder as the `TipCalculator.fxml` file you created in this section.

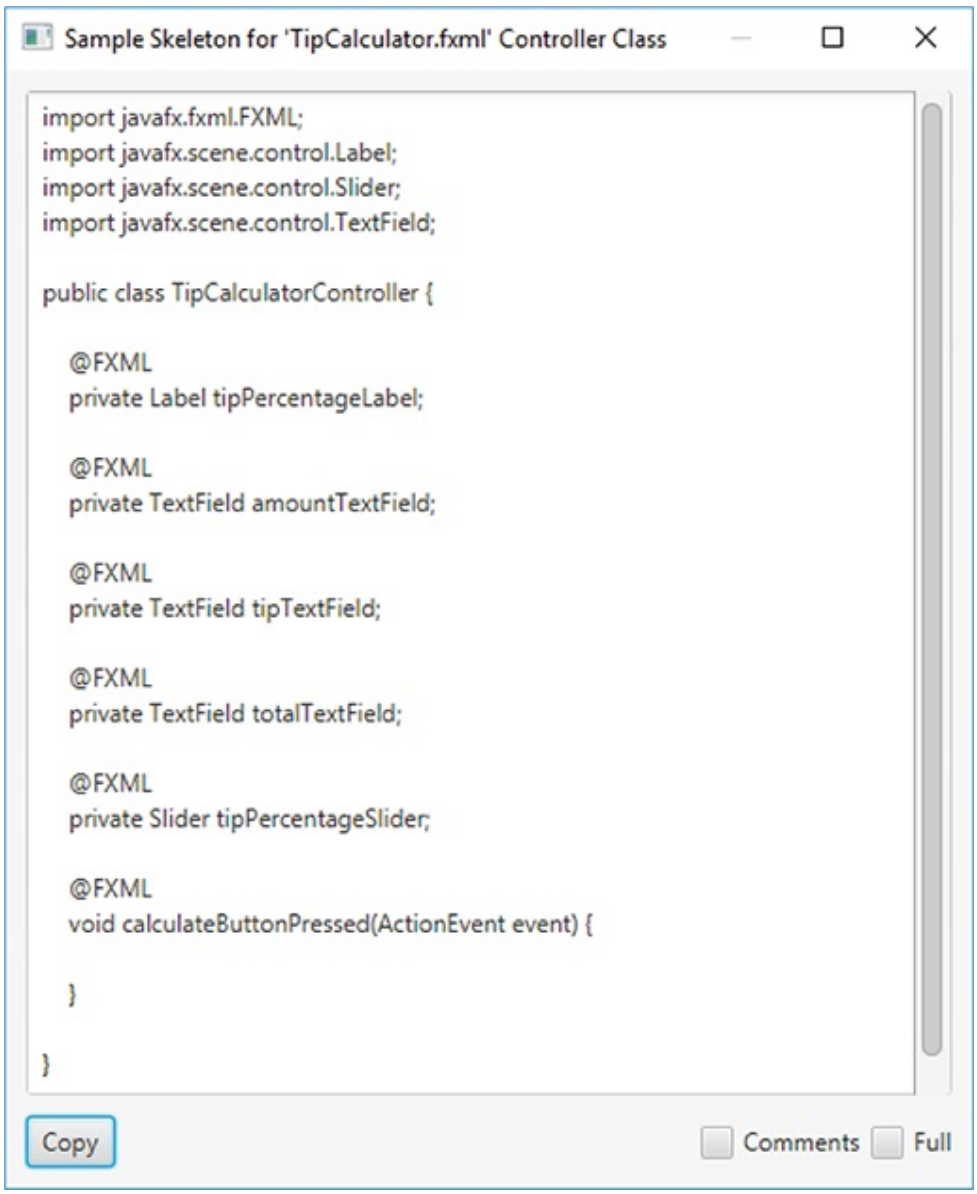


Fig. 12.18

Skeleton code generated by Scene Builder.

Description

12.5.4 `TipCalculator` Class

A simple JavaFX FXML-based app has two Java source-code files. For the **Tip Calculator** app these are:

- `TipCalculator.java`—This file contains the `TipCalculator` class (discussed in this section), which declares the `main` method that loads the FXML file to create the GUI and attaches the GUI to a `Scene` displayed on the app's `Stage`.
- `TipCalculatorController.java`—This file contains the `TipCalculatorController` class (discussed in Section 12.5.5), where you'll specify the `Slider` and `Button` controls' event handlers.

Figure 12.19 presents class `TipCalculator`. As we discussed in Section 12.5.2, the starting point for a JavaFX app is an `Application` subclass, so class `TipCalculator` extends `Application` (line 9). The `main` method calls class `Application`'s `static launch` method (line 23) to initialize the JavaFX runtime and to begin executing the app. This method causes the JavaFX runtime to create an object of the `TipCalculator` class and calls its `start` method (lines 10–19), passing the `Stage` object representing

the window in which the app will be displayed. The JavaFX runtime creates the window.

```
1   // Fig. 12.19: TipCalculator.java
2   // Main app class that loads and displays the Ti
3   import javafx.application.Application;
4   import javafx.fxml.FXMLLoader;
5   import javafx.scene.Parent;
6   import javafx.scene.Scene;
7   import javafx.stage.Stage;
8
9   public class TipCalculator extends Application {
10      @Override
11      public void start(Stage stage) throws Excepti
12          Parent root =
13              FXMLLoader.load(getClass().getResource(
14
15          Scene scene = new Scene(root); // attach s
16          stage.setTitle("Tip Calculator"); // displ
17          stage.setScene(scene); // attach scene to
18          stage.show(); // display the stage
19      }
20
21      public static void main(String[] args) {
22          // create a TipCalculator object and call
23          launch(args);
24      }
25  }
```

Fig. 12.19

Main app class that loads and displays the **Tip Calculator**'s GUI.

Overridden `Application` Method `start`

Method `start` (lines 11–19) creates the GUI, attaches it to a `Scene` and places it on the `Stage` that method `start` receives as an argument. Lines 12–13 use class `FXMLLoader`'s`static` method `load` to create the GUI's scene graph. This method:

- Returns a `Parent` (package `javafx.scene`) reference to the scene graph's root node—this is a reference to the GUI's `GridPane` in this app.
- Creates an object of the `TipCalculatorController` class that we specified in the FXML file.
- Initializes the controller's instance variables for the components that are manipulated programmatically.
- Attaches the event handlers specified in the FXML to the appropriate controls. This is known as registering the event handlers and enables the controls to call the corresponding methods when the user interacts with the app.

We discuss the initialization of the controller's instance variables and the registration of the event handlers in Section 12.5.5.

Creating the `Scene`

To display the GUI, you must attach it to a `Scene`, then attach the `Scene` to the `Stage` that method `start` receives as an argument. To attach the GUI to a `Scene`, line 15 creates a

`Scene`, passing `root` (the scene graph's root node) as an argument to the constructor. By default, the `Scene`'s size is determined by the size of the scene graph's root node. Overloaded versions of the `Scene` constructor allow you to specify the `Scene`'s size and fill (a color, gradient or image), which appears in the `Scene`'s background. Line 16 uses `Stage` method `setTitle` to specify the text that appears in the `Stage` window's title bar. Line 17 calls `Stage` method `setScene` to place the `Scene` onto the `Stage`. Finally, line 18 calls `Stage` method `show` to display the `Stage` window.

12.5.5 `TipCalculatorController` Class

Figure 12.20–12.23 present the `TipCalculatorController` class that responds to user interactions with the app's `Button` and `Slider`.

Class `TipCalculatorController`'s `import` Statements

Figure 12.20 shows class `TipCalculatorController`'s `import` statements.

```
1   // TipCalculatorController.java
2   // Controller that handles calculateButton and t
3   import java.math.BigDecimal;
4   import java.math.RoundingMode;
5   import java.text.NumberFormat;
6   import javafx.beans.value.ChangeListener;
7   import javafx.beans.value.ObservableValue;
8   import javafx.event.ActionEvent;
9   import javafx.fxml.FXML;
10  import javafx.scene.control.Label;
11  import javafx.scene.control.Slider;
12  import javafx.scene.control.TextField;
13
```

Fig. 12.20

`TipCalculatorController`'s `import` declarations.

The classes and interfaces used by class `TipCalculatorController` include:

- Class `BigDecimal` of package `java.math` (line 3) is used to perform precise monetary calculations. The `RoundingMode enum` of package `java.math` (line 4) is used to specify how `BigDecimal` values are rounded during calculations or when formatting floating-point numbers as `String`s.
- Class `NumberFormat` of package `java.text` (line 5) provides numeric formatting capabilities, such as locale-specific currency and percentage formats. For example, in the U.S. locale, the monetary value 34.95 is formatted as $34.95 and the percentage 15 is formatted as 15%. Class `NumberFormat` determines the locale of the system on which your app runs, then formats currency amounts and percentages accordingly.
- You implement interface `ChangeListener` of package

`javafx.beans.value` (line 6) to respond when the user moves the `Slider`'s thumb. This interface's `changed` method receives an object that implements interface `ObservableValue` (line 7)—that is, a value that generates an event when it changes.

- A `Button`'s event handler receives an `ActionEvent` object (line 8; package `javafx.event`) indicating which `Button` the user clicked. As you'll see in Chapter 13, many JavaFX controls support `ActionEvent`s.
- The annotation `FXML` (line 9; package `javafx.fxml`) is used in a JavaFX controller class's code to mark instance variables that should refer to JavaFX components in the GUI's FXML file and methods that can respond to the events of JavaFX components in the GUI's FXML file.
- Package `javafx.scene.control` (lines 10–12) contains many JavaFX control classes, including `Label`, `Slider` and `TextField`.

`TipCalculatorController`'s `static` Variables and Instance Variables

Lines 16—37 of Fig. 12.12 present class `TipCalculatorController`'s `static` and instance variables. The `NumberFormat` objects (lines 16–19) are used to format currency values and percentages, respectively. `NumberFormat` method `getCurrencyInstance` returns a `NumberFormat` object that formats values as currency using the default locale for the system on which the app is running. Similarly, `NumberFormat` method `getPercentInstance` returns a `NumberFormat` object that formats values as percentages using the system's default locale. The `BigDecimal` object `tipPercentage` (line 21)

stores the current tip percentage and is used in the tip calculation (Fig. 12.22) when the user clicks the **Calculate** `Button`.

```
14  public class TipCalculatorController {
15     // formatters for currency and percentages
16     private static final NumberFormat currency =
17        NumberFormat.getCurrencyInstance();
18     private static final NumberFormat percent =
19        NumberFormat.getPercentInstance();
20
21     private BigDecimal tipPercentage = new BigDec
22
23     // GUI controls defined in FXML and used by t
24     @FXML
25     private TextField amountTextField;
26
27     @FXML
28     private Label tipPercentageLabel;
29
30     @FXML
31     private Slider tipPercentageSlider;
32
33     @FXML
34     private TextField tipTextField;
35
36     @FXML
37     private TextField totalTextField;
38
```

Fig. 12.21

`TipCalculatorController`'s `static` and instance

variables.

@FXML Annotation

Recall from Section 12.5.3 that each control that this app manipulates in its Java source code needs an **fx:id**. Lines 24–37 (Fig. 12.21) declare the controller class's corresponding instance variables. The `@FXML` **annotation** that precedes each declaration (lines 24, 27, 30, 33 and 36) indicates that the variable name can be used in the FXML file that describes the app's GUI. The variable names that you specify in the controller class must precisely match the **fx:id** values you specified when building the GUI. When the `FXMLLoader` loads `TipCalculator.fxml` to create the GUI, it also initializes each of the controller's instance variables that are declared with `@FXML` to ensure that they refer to the corresponding GUI components in the FXML file.

`TipCalculatorController`'s `calculateButtonPressed` Event Handler

Figure 12.22 presents class `TipCalculatorController`'s

`calculateButtonPressed` method, which is called when the user clicks the **Calculate** `Button`. The `@FXML` annotation (line 40) preceding the method indicates that this method can be used to specify a control's event handler in the FXML file that describes the app's GUI. For a control that generates an `ActionEvent` (as is the case for many JavaFX controls), the event-handling method must return `void` and receive one `ActionEvent` parameter (line 41).

```
39      // calculates and displays the tip and total
                  40      @FXML
41      private void calculateButtonPressed(ActionEve
                 42         try {
43            BigDecimal amount = new BigDecimal(amou
44            BigDecimal tip = amount.multiply(tipPer
45            BigDecimal total = amount.add(tip);
                         46
47            tipTextField.setText(currency.format(ti
48            totalTextField.setText(currency.format(
                     49         }
  50         catch (NumberFormatException ex) {
51            amountTextField.setText("Enter amount")
    52            amountTextField.selectAll();
  53            amountTextField.requestFocus();
                    54         }
                      55      }
                          56
```

Fig. 12.22

`TipCalculatorController`'s

`calculateButtonPressed` event handler.

Registering the Calculate Button's Event Handler

When the `FXMLLoader` loads `TipCalculator.fxml` to create the GUI, it creates and registers an event handler for the **Calculate** `Button`'s `ActionEvent`. The event handler for this event must implement interface `EventHandler<ActionEvent>`—`EventHandler` is a generic type, like `ArrayList` (introduced in Chapter 7). This interface contains a `handle` method that returns `void` and receives an `ActionEvent` parameter. This method's body, in turn, calls method `calculateButtonPressed` when the user clicks the **Calculate** `Button`. `FXMLLoader` performs similar tasks for every event listener you specify via the Scene Builder **Inspector** window's **Code** section.

Calculating and Displaying the Tip and Total Amounts

Lines 43–48 calculate and display the tip and total. Line 43 calls the `amountTextField`'s `getText` method to get the bill amount typed by the user. This `String` is passed to `Big-Decimal`'s constructor, which throws a `NumberFormatException` if its argument is not a

number. In that case, line 51 calls `amountTextField`'s `setText` method to display the message `"Enter amount"` in the `TextField`. Line 52 then calls method `selectAll` to select the `TextField`'s text and line 53 calls `requestFocus` to give the `TextField` the focus. Now the user can immediately type a value in the `amountTextField` without having to first select its text. Methods `getText`, `setText` and `selectAll` are inherited into class `TextField` from class `TextInputControl` (package `javafx.scene.control`), and method `requestFocus` is inherited into class `TextField` from class `Node` (package `javafx.scene`).

If line 43 does not throw an exception, line 44 calculates the `tip` by calling method `multiply` to multiply the `amount` by the `tipPercentage`, and line 45 calculates the `total` by `add`ing the `tip` to the bill `amount`. Next lines 47 and 48 use the `currency` object's `format` method to create currency-formatted `String`s representing the tip and total amounts, which we display in `tipTextField` and `totalTextField`, respectively.

`TipCalculatorController`'s `initalize` Method

Figure 12.23 presents class `TipCalculatorController`'s `initialize` method.

This method can be used to configure the controller before the GUI is displayed. Line 60 calls the `currency` object's `setRoundingMode` method to specify how currency values should be rounded. The value `RoundingMode.HALF_UP` indicates that values greater than or equal to .5 should round up—for example, 34.567 would be formatted as 34.57 and 34.564 would be formatted as 34.56.

```
57      // called by FXMLLoader to initialize the con
        58      public void initialize() {
   59         // 0-4 rounds down, 5-9 rounds up
60         currency.setRoundingMode(RoundingMode.HALF
                    61
62         // listener for changes to tipPercentageSl
63         tipPercentageSlider.valueProperty().addLis
   64           new ChangeListener<Number>() {
           65              @Override
66              public void changed(ObservableValue<
67                 Number oldValue, Number newValue)
       68                 tipPercentage =
69                    BigDecimal.valueOf(newValue.in
70                 tipPercentageLabel.setText(percen
               71              }
                72            }
                 73         );
                   74      }
                     75   }
```

Fig. 12.23

`TipCalculatorController`'s `initalize` method.

Using an Anonymous Inner Class for Event Handling

Each JavaFX control has properties. Some—such as a `Slider`'s value—can generate events when they change. For such events, you must manually register as the event handler an object that implements the `ChangeListener` interface (package `javafx.beans.value`).

`ChangeListener` is a generic type that's specialized with the property's type. The call to `valueProperty` (line 63) returns a `DoubleProperty` (package `javax.beans.property`) that represents the `Slider`'s value. A `DoubleProperty` is an `ObservableValue<Number>` that can notify listeners when a value changes. Each class that implements interface `ObservableValue` provides method `addListener` (called on line 63) to register an event-handler that implements interface `ChangeListener`. For a `Slider`'s value, `addListener`'s argument is an object that implements `ChangeListener<Number>`, because the `Slider`'s value is a numeric value.

If an event handler is not reused, you often define it as an instance of an **anonymous inner class**—a class that's declared without a name and typically appears inside a method. The `addListener` method's argument is specified in lines 64–72 as one statement that

- declares the event listener's class,

- creates an object of that class and
- registers it as the listener for changes to the `tipPercentageSlider`'s value.

Since an anonymous inner class has no name, you must create an object of the class at the point where it's declared (thus the keyword `new` in line 64). A reference to that object is then passed to `addListener`. After the `new` keyword, the syntax

```
ChangeListener<Number>()
```

in line 64 begins the declaration of an anonymous inner class that implements interface `ChangeListener<Number>`. This is similar to beginning a class declaration with

```
public class MyHandler implements ChangeListener<Numb
```

The opening left brace at 64 and the closing right brace at line 72 delimit the anonymous inner class's body. Lines 65–71 declare the interface's `changed` method, which receives a reference to the `ObservableValue` that changed, a `Number` containing the `Slider`'s old value before the event occurred and a `Number` containing the `Slider`'s new value. When the user moves the `Slider`'s thumb, lines 68–69 store the new tip percentage and line 70 updates the `tipPercentageLabel`. (The notation `? extends Number` in line 66 indicates that the `ObservableValue`'s type argument is a `Number` or a subclass of `Number`. We

explain this notation in more detail in Section 20.7.)

Anonymous Inner Class Notes

8

An anonymous inner class can access its top-level class's instance variables, `static` variables and methods—in this case, the anonymous inner class uses the instance variables `tipPercentage` and `tipPercentageLabel`, and the `static` variable `percent`. However, an anonymous inner class has limited access to the local variables of the method in which it's declared—it can access only the `final` or effectively `final` (Java SE 8) local variables declared in the enclosing method's body.

Software Engineering Observation 12.2

The event listener for an event must implement the appropriate event-listener interface.

Common Programming Error 12.1

If you forget to register an event-handler object for a particular GUI component's event type, events of that type will be ignored.

Java SE 8: Using a Lambda to Implement the `ChangeListener`

8

Recall from Section 10.10 that in Java SE 8 an interface containing one method—such as `ChangeListener` in Fig. 12.23—is a functional interface. We'll show how to implement such interfaces with lambdas in Chapter 17.